Titre: Title:	Traçage noyau et distribué d'applications réparties pour l'analyse de la performance
Auteur: Author:	Loïc Gelle
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gelle, L. (2019). Traçage noyau et distribué d'applications réparties pour l'analyse de la performance [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3996/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Traçage noyau et distribué d'applications réparties pour l'analyse de la performance

LOÏC GELLE
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Traçage noyau et distribué d'applications réparties pour l'analyse de la performance

présenté par **Loïc GELLE**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Hanifa BOUCHENEB, présidente
Michel DAGENAIS, membre et directeur de recherche
Giuliano ANTONIOL, membre

DÉDICACE

À mes proches

REMERCIEMENTS

Je tiens avant tout à remercier le professeur Michel Dagenais qui a supervisé ce projet de recherche avec intérêt et bienveillance. Ses connaissances techniques sont aussi impressionnantes que la qualité de son enseignement et de sa gestion du DORSAL. Il a su donner du sens et de la consistance à mon projet de recherche et je lui en suis reconnaissant.

Un immense merci également à Geneviève Bastien pour sa grande disponibilité, sa bonne humeur, ses précieuses idées et sa grande connaissance de Trace Compass. Le DORSAL est chanceux de pouvoir compter sur toi.

Je remercie aussi tous mes collègues du DORSAL, en particulier Paul M., Robin et Yonni, qui m'ont accueilli et Arnaud, Guillaume et Paul N. qui ont rendu plus agréable la rédaction du présent mémoire.

Merci à Yuri Shkuro de Uber pour sa disponibilité et ses suggestions tout au long de ce projet, et à Ben Sigelman de Lightstep, qui a pris sur son précieux temps lors de KubeCon'18 pour me donner des conseils avisés.

Enfin, je remercie Ciena, EfficiOS, Ericsson, Google, le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) et Prompt pour leur soutien financier.

RÉSUMÉ

L'essor de l'infonuagique s'est accompagné de la transition depuis les applications monolithiques, lourdes et complexes à maintenir, vers les applications réparties sous forme de services flexibles et performants. Grâce à l'amélioration des outils de déploiement et de mise à l'échelle, ces dernières ont une efficacité accrue, une meilleure résistance aux pannes ainsi qu'un meilleur découpage sous forme de services développés indépendamment.

Toutefois, les applications réparties présentent des difficultés nouvelles liées à leur transparence de fonctionnement et à leur débogage. Une simple requête peut donner lieu à des dizaines de sous-requêtes interdépendantes et traitées par des machines physiques différentes. Le traçage distribué permet partiellement de répondre au problème en fournissant à l'utilisateur une vue en cascade de chaque requête au complet, incluant les durées de chaque sous-requête. C'est une technique précieuse qui permet d'identifier rapidement des requêtes défaillantes et des services anormalement lents. En revanche, et alors même que les applications ciblées sont conçues pour exécuter plusieurs requêtes en parallèle, le traçage distribué ne permet pas d'expliquer les causes d'une latence lorsque celle-ci découle d'une contention entre services ou entre requêtes concurrentes.

L'objectif de ce projet de recherche est d'étendre le traçage distribué à l'analyse de cause racine appliquée à des problèmes de contention. Notre hypothèse de travail est que, grâce à l'ajout d'événements de bas niveau issus de traces noyau, il est possible de créer des algorithmes permettant à l'utilisateur de comprendre la cause d'une latence observée dans une application distribuée. Nous proposons à cette fin de combiner le traçage distribué avec le traçage logiciel, de la collecte des événements à leur analyse ultérieure.

Nous avons conçu une méthode permettant de collecter et de synchroniser à la fois des événements de haut niveau, issus du traçage des requêtes, et des événements de bas niveau, issus du traçage du noyau. Notre technique repose sur l'instrumentation d'un traceur distribué par un traceur logiciel opérant dans l'espace utilisateur. Nous l'avons implémentée avec succès en utilisant respectivement les traceurs *Jaeger* et *LTTng*, et nous avons montré que l'impact sur la performance de l'application étudiée peut être limité par un faible échantillonnage des requêtes tracées.

Notre contribution principale est la conception d'un nouveau type d'analyse qui calcule, à partir des traces collectées précédemment, le chemin critique d'une requête donnée. Ce chemin critique est composé de tous les segments d'exécution, tels que vus par le noyau du système d'exploitation, qui contribuent au temps d'exécution total de la requête. Il permet de

visualiser les attentes et les blocages entre les requêtes, ou entre les requêtes et les ressources physiques de la machine.

Notre solution permet donc de visualiser et de mieux diagnostiquer les problèmes de performance d'une application répartie. Elle couvre les différentes étapes du débogage, de la collecte des traces d'exécution à leur analyse détaillée, le tout avec un surcoût temporel raisonnable et sans instrumentation additionnelle du code source de l'application visée.

ABSTRACT

Cloud applications have transitioned from centralized to distributed architectures deployed as smaller services. Tools for deploying and scaling distributed applications have made them more flexible, more efficient, more resilient and easier to maintain than their monolithic counterparts.

However, distributed applications are also more difficult to analyze and debug. Indeed, a single user request can trigger tens of subrequests than span across multiple services and physical machines. Distributed tracing partially addresses this problem by providing the user with the complete flow of a given request and the duration of each of its subrequests. While it is very useful for *identifying* faults and slow services, it fails at *explaining* the root cause when it is related to contention between services or requests. This is even more critical since distributed applications are often built for concurrency.

The objective of this research project is to extend distributed tracing to the root cause analysis of contention in distributed applications. Our assumption is that we can include low-level kernel events to trace analyses to better understand the cause of detected latency. To that end, we propose combining distributed tracing with kernel tracing, from collecting to analyzing the events.

We designed a technique for collecting and synchronizing high-level distributed tracing events and low-level kernel events altogether. Our solution is based on instrumenting a distributed tracer with a software tracer that operates in user space. We used Jaeger as a distributed tracer and LTTng as a software tracer for our implementation. We showed that the performance impact of our solution could be controlled by lowering the sampling frequency of tracing in the target application.

Our main contribution is the design and implementation of a new trace analysis that combines distributed and kernel traces. It outputs the critical path of a target request, that is the succession of the kernel states that contribute to the total execution time of that request. The analysis helps visualize the waiting and blocking dependencies between the different requests, or between the requests and the physical resources of the machine.

We argue that our solution improves the visualization and the diagnosis of performance issues in distributed applications. We cover the whole debugging process, from trace collection to analysis and visualization. Most importantly, our solution has a reasonable overhead and requires no added instrumentation of the target application.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface (interface de programmation applicative)
BPF	Berkeley Packet Filter
CPU	Central Processing Unit (processeur)
CTF	Common Trace Format
DNS	Domain Name Service
eBPF	Extended Berkeley Packet Filter
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
PID	Process Identifier
RAM	Random Access Memory
RPC	Remote Procedure Call (appel de procédure à distance)
TID	Thread Identifier
TLB	Translation Lookaside Buffer
TLS	Thread-Local Storage (mémoire local de thread)

CHAPITRE 1 INTRODUCTION

Les applications réparties, plus particulièrement sous forme de microservices, sont rendues de plus en plus populaires dans les déploiements infonuagiques. Si elles offrent généralement une plus grande performance, une meilleure résilience et une flexibilité accrue par rapport aux applications monolithiques, elles sont aussi plus difficiles à déboguer puisqu'une même transaction utilisateur peut être traitée par plusieurs machines physiques.

Le traçage distribué répond à ce défi en propageant tout au long de la transaction le contexte nécessaire à la reconstruction ultérieure des requêtes utilisateur. Les traceurs offrent la possibilité d'isoler les requêtes qui ont pris un temps inhabituel et d'afficher le chemin suivi en fonction du temps par la transaction, ainsi que les liens de parenté entre les différentes sous-requêtes. Cela permet d'obtenir un profil des requêtes les plus fréquentes ou de détecter assez facilement des problèmes de logique dans l'application. En revanche, le traçage distribué ne permet pas d'expliquer des problèmes plus complexes qui sont liés à la manière dont l'application interagit avec le système hôte.

Le traçage noyau, d'un autre côté, est spécifiquement dédié à la collecte et à l'analyse d'événements de bas niveau. Les analyses les plus complexes permettent de reconstruire le chemin critique d'un fil d'exécution donné et d'en déduire les causes racines des latences observées.

Dans ce travail, nous nous intéressons à la combinaison des techniques de traçage distribué et de traçage noyau afin de mieux caractériser la performance des applications distribuées.

1.1 Définition des concepts de base

Dans cette section, nous définissons les concepts qui seront utiles à la compréhension de ce mémoire, et notamment de la revue de littérature.

1.1.1 Application distribuée

Une application distribuée est un ensemble de services, communiquant par le biais de messages pour la réalisation d'une tâche complexe. Elle s'oppose à une application monolithique qui consiste en un unique programme exécuté sur une machine. Les applications distribuées sont rendues populaires par le développement d'intergiciels de communication standardisés simplifiant l'appel de procédures à distance. Pour peu que chaque service expose une interface de programmation applicative claire, une application distribuée est plus simple à maintenir, puisque des services différents – base de données, calcul, authentification, etc. – peuvent être

développés, répartis et mis à l'échelle séparément.

De la même manière qu'une application classique, chaque service d'une application distribuée peut produire des fichiers journaux, dans lesquels sont inscrits des événements lors de l'exécution selon ce qui a été défini par les développeurs. Ces événements peuvent ensuite être lus par les développeurs eux-mêmes, ou des administrateurs du système, afin de surveiller le bon fonctionnement de l'application et de comprendre la cause d'éventuels dysfonctionnements. Une limitation des fichiers journaux est que les événements qui y sont inscrits sont définis a priori. De plus, afin de conserver une taille raisonnable des fichiers journaux, seuls les événements essentiels y sont inscrits et il n'est donc pas toujours possible de disposer de l'information nécessaire pour reproduire un bogue.

1.1.2 Traçage logiciel

Le traçage logiciel est une technique visant à collecter des événements sur un système de manière exhaustive et performante. Il peut concerner aussi bien des applications utilisateur qui sont instrumentées pour produire des événements pertinents que d'autres composants du système comme le noyau. Le traçage logiciel repose sur des points de trace, insérés statiquement dans le code source d'une application ou du noyau, qui émettent un événement à chaque fois qu'ils sont atteints. Le traceur a pour rôle de collecter les événements émis avec un impact minimal sur la performance du système, puis de les inscrire dans des fichiers de trace pour qu'ils soient analysés ultérieurement. Le noyau Linux est instrumenté par défaut avec un grand nombre de points de trace statiques qui permettent par exemple d'émettre un événement à chaque appel système traité.

Un fichier de trace contient tous les événements qui ont été enregistrés pendant le traçage, dans l'ordre chronologique. Un événement contient une estampille de temps précise, et généralement plusieurs champs définis lors de l'ajout du point de trace – par exemple la valeur de retour et les arguments dans le cas d'un appel système. Il peut être accompagné d'un contexte utile à l'analyse, par exemple l'identifiant du fil d'exécution qui a émis l'événement, ou l'identifiant du cœur de processeur qui a atteint le point de trace.

La taille des fichiers de trace ainsi que le niveau d'abstraction des événements qui y sont inscrits nécessitent de faire appel à des outils spécialisés d'analyse. Ces outils sont capables de parcourir efficacement la trace pour calculer l'évolution de métriques pertinentes, retrouver l'état précis du système à un instant donné ou encore chercher des anomalies lors de l'exécution.

1.1.3 Chemin critique d'une application

Le chemin critique d'un fil d'exécution est la plus longue suite d'opérations qui sont exécutées de manière séquentielle – et non en parallèle – par ce fil d'exécution, ou par d'autres fils d'exécution desquels il dépend. Dans le cas de fils d'exécution qui interagissent indirectement par le biais d'une ressource partagée, il est tout à fait possible que leurs chemins critiques soient entrelacés – autrement dit, certaines opérations du premier fil d'exécution sont dans le chemin critique du second, et vice-versa. Le chemin critique permet ainsi de se rendre compte rapidement des dépendances indirectes qui peuvent provoquer une contention entre différents fils d'exécution. Les ressources sujettes à contention entre différents fils d'exécution sont usuellement l'accès au disque ou au réseau, ou l'accès à une primitive d'exclusion mutuelle comme un *mutex*.

Dans la suite de ce mémoire, nous utiliserons également la notion de chemin critique d'une application. Pour nos besoins, il s'agira du chemin critique du fil d'exécution dans lequel l'application s'exécute. Pour une application qui s'exécute dans une succession de fils d'exécution, le chemin critique de l'application est la succession des chemins critiques de ces mêmes fils d'exécution.

1.2 Éléments de la problématique

Avec l'utilisation grandissante des applications distribuées dans le nuage vient la nécessité de concevoir des outils permettant de les déboguer. Nous distinguons deux problématiques centrales.

(1) Les paradigmes émergents de développement et de déploiement de ces applications encouragent le recours à des couches d'abstraction de plus en plus nombreuses :

- les conteneurs et les machines virtuelles permettent d'isoler et de limiter les ressources consommées par une application, tout en facilitant la mise à l'échelle sur de multiples serveurs ;
- les librairies de communication, d'appel de procédure à distance et de gestion des tâches asynchrones facilitent le développement et la standardisation des applications ;
- les langages de programmation offrent des fonctionnalités comme la récupération dynamique de l'espace mémoire ou la gestion dans l'espace utilisateur de fils d'exécution légers.

Ces couches d'abstraction facilitent le développement, la maintenance et le déploiement des applications au détriment de leur transparence de fonctionnement. Sachant qu'une

partie non négligeable de la pile logicielle qui s'exécute n'a pas été écrite par le développeur de l'application, le débogage est rendu plus difficile et doit faire appel à des outils spécialisés.

(2) Les différents éléments d'une application répartie – base de données, interface client, etc. – peuvent être distribués sur des serveurs différents et communiquent au moyen de messages. Le défi est de réussir à retracer le chemin suivi par une requête utilisateur lorsqu'elle s'étend sur plusieurs services et machines physiques, notamment parce que les différents fichiers journaux ne sont pas forcément centralisés. Il est en effet utile de savoir dans le cadre de quelle requête utilisateur une anomalie a pu survenir, de manière à pouvoir reproduire puis résoudre le bogue. Tracer la séquence d'opérations survenue pendant une transaction permet de plus de faire du profilage, afin de savoir quel service de l'application est le plus lent à répondre.

Le traçage distribué – à distinguer du traçage logiciel défini plus haut – permet de répondre au deuxième point soulevé. Il permet d'instrumenter les intergiciels de communication afin de retracer la séquence complète d'opérations de chaque requête utilisateur. Chaque transaction se voit attribuer un identifiant unique, qui est injecté dans les messages échangés par les services afin d'être propagé dans les sous-transactions. Le traceur présent sur chaque machine est chargé de transmettre les événements au collecteur central qui sauvegarde les traces. L'utilisateur est ainsi en mesure de visualiser les traces en faisant des requêtes au collecteur. Le traçage distribué gagne en popularité, notamment grâce à l'initiative de spécification en source ouverte *OpenTracing*. Un exemple de trace distribuée est donné dans l'article en figure 4.1.

Le traçage distribué ne permet en revanche pas de résoudre le premier point soulevé. Les événements enregistrés dans les traces distribuées sont de haut niveau et ne montrent pas les interactions entre les différentes requêtes, ou entre les services et le noyau du système d'exploitation. Ils permettent généralement de détecter des problèmes de logique dans l'application distribuée – mauvais chemin suivi par une requête, arguments erronés dans un message, parallélisation insuffisante – mais pas d'expliquer pourquoi une sous-requête en particulier a duré anormalement longtemps. Les sources potentielles de contention sont pourtant nombreuses pour une requête, par exemple :

— la compétition entre requêtes ou processus pour accéder au CPU ;

— l'attente après un verrou partagé entre plusieurs fils d'exécution ;

— l'attente après l'accès au disque ou au réseau ;

— une configuration trop restrictive des limites de ressources imposées à l'application.

Le traçage logiciel du noyau paraît être un bon candidat pour résoudre le premier point. Le nombre et la précision des événements collectés, ainsi que la complexité des analyses proposées par les outils dédiés, permettent en pratique une analyse de cause racine efficace. En revanche, les traceurs logiciels ne sont pas conçus pour propager le contexte d'une requête tout au long d'une transaction, ou pour être déployés facilement dans un système réparti. Ils répondent à des critères très précis de performance et sont pensés pour perturber le moins possible le comportement du système ciblé. Ce n'est pas envisageable pour un traceur logiciel de sacrifier la performance au profit de la capacité à tracer des transactions distribuées.

Il est cependant possible d'étudier des approches hybrides. Le but de notre travail est de combiner les avantages du traçage noyau et du traçage distribué afin de mieux caractériser la performance des applications distribuées. Nous utiliserons en particulier le traceur *LTTng* pour Linux, dont la pertinence du choix est justifiée dans la revue de littérature, pour recueillir les événements de bas niveau requis pour les analyses. L'idée générale est d'instrumenter un traceur distribué avec *LTTng* afin de synchroniser les événements de la trace distribué avec les événements de la trace *LTTng*. Cette synchronisation doit nous permettre d'analyser par la suite conjointement les traces distribuées avec les traces noyau, dans le but de comprendre plus en détail la performance d'une application répartie.

1.3 Objectifs de recherche

Notre objectif principal est de proposer des outils de collecte et d'analyse de traces qui permettent de comprendre de manière précise la performance des requêtes d'une application distribuée.

La première étape de notre travail consistera en la conception d'un mécanisme de collecte de trace adapté aux applications distribuées. Il s'agira de collecter à la fois les traces distribuées à l'aide d'un traceur adapté et les traces noyau à l'aide de *LTTng*, de manière à ce que les deux traces puissent être analysées conjointement. En particulier, il faudra s'assurer que les événements de la trace distribuée se voient attribuer une estampille de temps suffisamment précise pour que l'analyse en soit possible. Nous veillerons également à ce que le surcoût de la collecte des traces soit suffisamment faible pour ne pas perturber de manière significative le comportement de l'application. Dans un second temps, nous proposerons une analyse conjointe des traces qui permette de comprendre l'interaction de n'importe quelle requête distribuée avec le système hôte, ses ressources et ses autres processus. Cette analyse doit permettre de valider l'approche utilisée pour la collecte des traces. Les résultats de ces analyses pourront être visualisés dans l'outil d'analyse de traces *Trace Compass*.

Dans ce mémoire, nous chercherons donc à répondre aux questions de recherche suivantes :

Q1. Comment collecter conjointement des traces noyau et des traces distribuées, sans perturber de manière significative le comportement de l'application ?

Q2. Comment synchroniser les deux types de traces dans un outil classique d'analyse de traces ?

Q3. Comment produire une analyse transversale qui utilise à la fois les informations de haut niveau des traces distribuées et les informations de bas niveau des traces noyau ?

Q4. Comment afficher les résultats d'une telle analyse afin qu'ils soient utiles à un utilisateur, à des fins de débogage et de compréhension de la performance de l'application ?

1.4 Plan du mémoire

Le chapitre 2 est une revue critique de la littérature, qui présente l'état de l'art dans les domaines de l'analyse de la performance des applications distribuées d'une part, et du traçage logiciel d'autre part. Le chapitre 3 détaille la méthodologie utilisée pour le développement de ce projet et l'obtention des résultats.

Le chapitre 4 constitue le cœur de ce mémoire. Il s'agit d'un article qui décrit la solution proposée et qui discute les résultats obtenus.

Le chapitre 5 complète et approfondit la discussion présentée dans l'article, et le chapitre 6 conclut ce mémoire en récapitulant le travail effectué.

CHAPITRE 2 REVUE CRITIQUE DE LA LITTÉRATURE

Dans ce chapitre, nous présenterons les travaux effectués précédemment dans les domaines reliés à notre travail. Nous nous intéresserons dans un premier temps aux techniques d'analyse de performance des applications réparties, et dans un second temps aux techniques et outils de traçage logiciel.

2.1 Analyse de la performance des applications réparties

De nombreuses techniques ont été étudiées en vue d'analyser la performance d'applications réparties. Pour les besoins de cette revue de littérature, nous allons distinguer les approches selon qu'elles utilisent :

— Des techniques *ad hoc* d'instrumentation et d'analyse des applications réparties ;

— Le traçage distribué, une technique rendue populaire par l'outil *Dapper* de Google (Sigelman et al., 2010) ;

— Un mélange des deux techniques ci-dessus ; autrement dit, il s'agit de travaux utilisant le traçage distribué pour instrumenter les applications, mais qui utilisent des techniques *ad hoc* pour analyser les données collectées.

2.1.1 Techniques *ad hoc*

Dans cette section, nous présenterons les travaux par ordre croissant de spécificité : les premières techniques présentées sont les plus globales, soit parce que leur portée est large, soit parce qu'elles ne nécessitent que peu de modifications du système étudié.

Approches globales portant sur l'impact de la configuration de l'application

Une première approche consiste à mesurer de manière systématique l'impact que peut avoir le contexte d'une application sur son bon fonctionnement. Un travail de Chen et al. (2009) combine par exemple des processus de décision markoviens avec des techniques d'apprentissage machine pour déterminer après quelques essais la configuration optimale – en matière de performance – d'une application donnée. Appliquée à l'exemple du serveur web Apache, la technique permet d'augmenter l'efficacité du serveur d'environ 20% sans aucune connaissance de l'architecture de l'application. Cela est particulièrement utile pour utiliser au mieux les ressources fournies par les grappes de serveur dont on dispose. En analysant une trace

d'exécution gigantesque rendue publique par Google – 29 jours de données pour une taille compressée de 39 gigaoctets – sur une grappe de serveurs, Reiss et al. (2012) constatent que l'ordonnanceur de tâches utilisé a tendance à surestimer la charge requise pour les applications et que les serveurs ne sont utilisés au mieux qu'à 40-60% de leur capacité. Les ordonnanceurs ont tendance à ne pas planifier deux tâches sensibles à la latence sur une même machine afin de préserver la qualité de service, quitte à sous-utiliser les grappes de calcul. Afin de résoudre ce dilemme, Leverich et Kozyrakis (2014) étudient des politiques d'ordonnancement plus flexibles qui permettent d'utiliser au mieux les ressources des machines sans sacrifier la qualité de service. L'idée est de partager les ressources d'une machine entre des tâches très sensibles à la latence et des tâches qui le sont moins, afin d'augmenter l'utilisation globale des serveurs. Les auteurs montrent que les algorithmes qu'ils ont conçus peuvent augmenter de 52% le rapport entre le débit de requêtes et le coût de l'infrastructure dans le meilleur des cas.

Dans le même ordre d'idées, l'injection de faute contrôlée est une technique qui permet de vérifier la robustesse d'une application et d'estimer sa performance dans le pire des cas ; un outil comme *Chaos Monkey* (Bennett et Tseitlin, 2012), développé par Netflix, a rendu populaire le paradigme de *Chaos Engineering* (Basiri et al., 2016) – ou "ingénierie du chaos" – pour tester la résilience des applications réparties. L'injection de faute contrôlée permet également de créer une base de données de fichiers journaux correspondant à des exécutions dont on connaît les défaillances ; il est alors possible d'utiliser ces données pour "deviner" la cause racine d'une défaillance inconnue en comparant les fichiers journaux, produits par l'application, à la base de données (Pham et al., 2017).

Identification des causes racines sans modifications du code source ou de l'exécutable

Les approches citées ci-dessus ne sont pas pertinentes dans le cadre de notre projet, qui vise à comprendre la cause racine des latences observées plutôt que les éliminer systématiquement. Les travaux qui sont présentés par la suite s'intéressent tous d'une certaine manière à l'analyse de la cause racine des latences, ou au moins à l'identification des chemins d'appels les plus coûteux d'une application.

L'intérêt a toujours été grand d'obtenir des méthodes peu intrusives d'analyse de la performance. Beaucoup de travaux mettent d'ailleurs en avant le fait que leur méthode ne requiert pas ou peu d'instrumentation de l'application visée ou des librairies de communication entre processus. Ces méthodes s'appuient en grande majorité sur l'analyse des fichiers journaux produits par toutes les composantes d'une application répartie. On peut ainsi obtenir des

résultats satisfaisants avec une simple collecte des fichiers journaux, qui sont ensuite analysés pour reconstruire les chemins d'appels d'une application et déterminer quels sont les chemins les plus coûteux (Aguilera et al., 2003). De la même manière, l'outil *lprof* de Zhao et al. (2014) est capable d'analyser statiquement les fichiers binaires de l'application cible pour comprendre comment interpréter correctement les fichiers journaux ; l'outil permet ensuite de reconstruire chaque requête utilisateur séparément à partir de ces seuls fichiers journaux. L'approche a été validée avec succès sur des applications réparties populaires telles que HDFS, Yarn ou encore Cassandra. Enfin, *Synoptic* (Beschastnikh et al., 2011) est un outil capable de construire un modèle correspondant à une application et qui respecte ses invariants de temps, en utilisant seulement les fichiers journaux produits par cette application. Le développeur doit seulement fournir des expressions régulières qui permettent d'extraire les informations des fichiers journaux. Le modèle généré permet de détecter rapidement des bogues de logique dans l'application, et présente l'avantage d'être facilement lisible sans pour autant connaître le code source de l'application.

L'apprentissage machine a aussi été mis à profit pour analyser de grandes quantités de fichiers journaux dans le but de détecter et de classifier des problèmes de performance. On peut par exemple superviser l'apprentissage en séparant les fichiers journaux issus d'exécutions performantes de fichiers journaux problématiques (Nagaraj, Killian et Neville, 2012). L'outil *Draco* de Kavulya et al. (2012) est capable d'identifier plus précisément des "signatures" correspondant à des problèmes de performance spécifiques, en combinant des techniques d'apprentissage machine avec la mesure de divergence d'entropie relative. L'outil est ainsi capable dans 97% des cas observés en production de déterminer correctement la cause de la latence ; il peut s'agir de problèmes de configuration, de contention sur le serveur ou encore d'un défaut logiciel. Enfin, l'outil *vPath* de Tak et al. (2009) utilise également une approche passive de collecte des données pour reconstruire les requêtes à partir des fichiers journaux. Un moniteur est chargé d'inscrire dans un fichier journal tous les événements d'envoi et de réception liés aux connexions TCP ouvertes par les fils d'exécution ciblés. L'analyse peut ensuite se servir de ces événements pour caractériser a posteriori la performance de l'application.

L'interposition de librairie est une technique qui permet d'obtenir des informations plus précises que celles fournies par les fichiers journaux d'une application non modifiée, sans pour autant avoir besoin d'instrumenter l'application en question. L'idée est de charger, au lancement d'une application, une librairie qui remplacera partiellement les librairies standards utilisées par l'application. Cela est particulièrement utile si on veut modifier le comportement de certaines fonctions des librairies largement utilisées comme `libc`. *Borderpatrol* (Koskinen et Jannotti, 2008), en particulier, met a profit l'interposition de librairie pour collecter des

traces d'exécution dans des applications réparties et sans aucune instrumentation. Un petit nombre – une vingtaine – de fonctions de bas niveau de la librairie `libc` sont interceptées afin de noter les événements correspondant à des opérations d'écriture, d'ouverture d'un tube de communication ou encore de réception d'un message sur un socket. En traçant tous les événements de communication à la frontière des composantes de l'application – d'où le nom de *Borderpatrol*, l'outil est capable de reconstruire le flot de chacune des requêtes. Le surcoût n'est pas négligeable et est estimé à environ 10-15% en moyenne, mais il est très variable selon la nature de la tâche effectuée. Pour un serveur Apache dont les requêtes sont étudiées pour provoquer beaucoup de création de processus, le surcoût peut dépasser les 300%, car la librairie de traçage doit s'attacher dynamiquement à tous les processus fils créés. Cela limite la solution proposée pour les applications qui sont gourmandes en appels système.

Identification des causes racines avec modification du code source ou de l'exécutable

Contrairement aux travaux évoqués ci-dessus, les techniques qui suivent nécessitent d'avoir accès au code source de l'application cible et de l'instrumenter plus ou moins directement.

Une première approche qui met à profit l'accès au code source sans requérir pour autant l'écriture de code dédié à l'instrumentation est l'annotation du code source. Une annotation consiste en l'écriture d'un commentaire qui ne sera pas directement traduit en code compilé par le compilateur, mais qui pourra être interprété par d'autres outils – par exemple des profileurs – pour aider à la compréhension du code source. Dans le domaine de la sécurité informatique, par exemple, il est utile de savoir quelles actions de l'utilisateur dans un navigateur Internet ont pu conduire au téléchargement d'un logiciel malveillant. Ma et al. (2017) ont proposé une technique à base d'annotation de code pour intégrer des informations sur des actions de l'utilisateur – ouverture d'un onglet, clic sur un lien, etc. – aux fichiers journaux produits des applications telles que Firefox ou Mplayer. Les annotations sont lues par le compilateur qui les transforme en instructions d'écriture dans les fichiers journaux, pour un surcoût observé lors de l'exécution d'environ 1% dans le pire des cas. L'outil *Pip* de Reynolds et al. (2006) utilise une approche similaire pour aider à la détection de bogues et de problèmes de performance dans des applications distribuées. Cette fois-ci, les annotations permettent aux développeurs de préciser le comportement attendu sous forme d'assertions pouvant porter par exemple sur la latence d'une requête. L'outil permet ensuite de visualiser les fichiers journaux correspondant à des exécutions ne respectant pas les contraintes fixées par les annotations. Enfin, *vProfiler* (Huang, Mozafari et Wenisch, 2017) permet aux développeurs d'identifier les sections critiques du code qui correspondent à des tâches de

haut niveau et dont on veut évaluer la durée. L'outil fournit de plus une instrumentation des primitives de synchronisation et de communication pour reconstruire le chemin critique d'une tâche. Il a été appliqué avec succès à des systèmes comme MySQL ou Apache et donne pour chaque tâche identifiée le détail du temps passé par fonction dans le code.

D'autres approches requièrent de modifier activement l'application cible. Pour limiter les modifications à apporter au code source, une approche consiste à instrumenter dynamiquement l'application, c'est-à-dire à ajouter des instructions de débogage alors que l'instruction est en cours d'exécution. Le code est modifié en mémoire, sans avoir besoin de recompiler ou de redémarrer l'application. L'intérêt est particulièrement grand pour instrumenter des applications déployées en production et qui ont des contraintes de disponibilité. L'outil *X-ray* de Attariyan, Chow et Flinn (2012) met à profit cette approche dans le but d'analyser la cause racine de défauts de performance dans des applications en production. L'utilisateur doit spécifier un filtre sous forme d'expression régulière qui permet de détecter les paquets réseau entrants correspondant à un début de requête ; chaque requête est ensuite suivie grâce aux événements collectés, par exemple par l'instrumentation dynamique des appels système pertinents et des opérations de synchronisation. *X-ray* permet donc d'extraire les requêtes qui prennent un temps inhabituel et donne la durée d'exécution de chaque bloc de code de l'application chargé de traiter la requête. La technique permet donc de déterminer la cause racine de la latence dans le code directement. Dans un registre différent, *Fay* (Erlingsson et al., 2012) utilise des techniques d'instrumentation dynamique à des fins de monitorage. L'utilisateur peut définir des métriques arbitraires qui sont ensuite collectées en insérant dynamiquement du code d'instrumentation dans les composantes de l'application cible. Plus généralement, les techniques d'instrumentation dynamique présentent un intérêt certain dans l'analyse d'applications réparties et seront donc développées un peu plus tard dans cette revue de littérature.

En résumé

- Nous nous sommes intéressés à des techniques d'analyse de la performance des applications réparties qui n'impliquent pas de collecte active d'informations sur les requêtes. Il s'agit usuellement d'analyser les fichiers journaux non modifiés produits par une application pour en extraire des informations sur son comportement, sa logique, sa latence, et éventuellement ses anomalies. Cette approche n'est pas suffisamment précise pour nos besoins et elle nécessite un grand nombre de fichiers journaux pour être efficace.
- De plus en plus de travaux s'intéressent à l'analyse de la cause racine des la-

tences détectées dans des transactions réparties. À ce stade, les approches sont assez différentes les unes des autres et nous n'avons pas mis en lumière un effort de standardisation. On note tout de même que les techniques les plus précises nécessitent soit une instrumentation des intergiciels, soit l'annotation du code source de l'application.

— L'instrumentation dynamique est rendue de plus en plus populaire pour le suivi des applications en production. Le principal intérêt de la technique réside dans le fait que le code source des applications n'a pas besoin d'être modifié ou recompilé.

2.1.2 Le traçage distribué

Le traçage distribué s'intéresse à la collecte d'informations de trace de bout en bout dans une application distribuée. Il permet de suivre chaque requête d'un utilisateur dans son traitement par chacune des composantes de l'application. Dans la collecte d'informations, le traçage distribué se distingue des approches précédentes par :

— l'instrumentation explicite du code des applications et des intergiciels ;
— l'accent qui est mis sur la propagation du contexte de traçage d'un bout à l'autre de l'application ;
— la standardisation progressive du domaine par le biais d'interfaces de programmation ;
— l'émergence de traceurs en source libre dont l'architecture et le déploiement sont adaptés aux applications réparties.

Dans cette section, nous présenterons les outils pertinents en suivant l'ordre chronologique. Cela permettra de suivre l'évolution du traçage distribué, depuis les premiers travaux jusqu'à l'état actuel de l'art (Sambasivan et al., 2016).

Premiers travaux

Magpie (Barham et al., 2004) est sans doute un des outils précurseurs dans le domaine du traçage distribué. Il n'est pas aussi complet que les outils actuels mais propose déjà un suivi des requêtes par l'instrumentation ciblée de certains sous-systèmes du noyau de Windows NT. On retrouve déjà dans ce travail tous les éléments importants à considérer lorsqu'on veut tracer une application distribuée :

— la synchronisation des traces lorsqu'elles proviennent de machines différentes, dont les horloges ne sont pas nécessairement synchronisées ;

— le besoin d'identifier les événements qui sont causalement liés afin de proposer une vue cohérente de chaque requête de l'utilisateur ;

— la contrainte de charge qui doit limiter l'impact du traçage sur la performance de l'application.

Magpie utilise l'infrastructure de traçage *Event Tracing for Windows* qui est fournie avec Windows NT. Le travail souligne l'importance de tracer les événements de bas niveau correspondant à l'envoi ou à la réception de RPC, car ce sont les événements qui permettront de reconstruire la requête lorsqu'elle se propage à différentes composantes de l'application distribuée. *Event Tracing for Windows* (Microsoft, 2018) permet également de récolter des événements au niveau du noyau concernant l'exécution et la synchronisation des fils d'exécution, l'accès au disque et au réseau ou encore la consommation de temps CPU. Après analyse de la trace, ces événements permettent de rendre compte de l'impact que chacune des requêtes a pu avoir sur d'une part la charge globale du système, et d'autre part l'état (en cours d'exécution ou bloqué) de chacun des fils d'exécution impliqués dans la requête. Cette volonté d'associer de tels événements de bas niveau à des informations de plus haut niveau concernant des requêtes d'utilisateurs est à souligner ; même si le contexte est différent et que les technologies ont évolué, l'idée reste une composante majeure du travail présenté dans ce mémoire.

Quelques années plus tard, *X-Trace* (Fonseca et al., 2007) a précisé l'approche en posant des principes architecturaux pertinents pour le traçage de systèmes répartis complexes :

— l'information nécessaire au traçage – un identifiant unique de requête notamment – doit suivre le même chemin que la requête afin d'obtenir une trace de bout en bout fidèle à l'exécution réelle ;

— au contraire, l'information de trace collectée doit suivre un chemin de données différent afin de (i) minimiser la charge sur la bande passante allouée pour les requêtes et (ii) pouvoir récolter les événements même en cas de problème réseau sur le chemin de la requête ;

— les données collectées lors du traçage ne sont rendues disponibles qu'aux utilisateurs qui en ont besoin et qui en ont les permissions ; cela est particulièrement utile si les données collectées contiennent des informations sensibles sur les utilisateurs.

Ces principes seront repris par les travaux et outils suivants dans le domaine du traçage distribué. *X-Trace* définit également un format de métadonnées capable d'encapsuler toutes les informations pertinentes pour reconstruire le chemin de la requête *a posteriori*, à savoir (i) un identifiant de requête unique, (ii) l'identifiant de la requête parente et (iii) l'identifiant

de la machine à laquelle envoyer l'événement de trace une fois la requête traitée. C'est à l'utilisateur qui instrumente son application d'injecter et d'extraire les métadonnées dans et depuis les messages échangés entre les composantes de l'application. Cela est plus ou moins facile selon le protocole utilisé ; *X-Trace* propose dans sa librairie de traçage le support, entre autres, des protocoles HTTP, IP et DNS.

L'outil Dapper de Google

Si les principes posés par *X-Trace* sont intéressants, l'article manque tout de même d'une évaluation complète de l'outil, notamment au niveau de la taille des traces générées et du surcoût de performance engendré par le traçage. L'outil *Dapper* (Sigelman et al., 2010) de Google est accompagné d'une évaluation plus complète de son impact sur la performance des applications instrumentées. De plus, même si beaucoup d'éléments sont communs avec *X-Trace*, certains choix architecturaux ont été faits pour que l'outil puisse permettre aux développeurs d'analyser la performance de leurs applications à l'échelle de Google.

Choix architecturaux L'article parle de "ubiquitous development" pour souligner cette volonté d'intégrer l'analyse de la performance de l'application distribuée entière lors du développement. Pour une simple recherche sur Google, la requête peut être traitée par des milliers de machines différentes et plusieurs services pour chercher des résultats de natures différentes – images, vidéos, actualité, etc. Cela est donc particulièrement important que chaque service puisse être instrumenté afin de pouvoir optimiser d'abord le service le plus lent, après lequel l'utilisateur devra attendre. Les services évoluant rapidement en interne, il faut que chaque équipe puisse déterminer quel outil – potentiellement maintenu par une équipe différente – est responsable d'une latence nouvellement observée dans leur produit. Les concepteurs de l'outil souhaitaient aussi s'affranchir d'une instrumentation manuelle propre à chaque application, qui aurait été plus difficile à coordonner entre les équipes. *Dapper* s'appuie donc sur l'instrumentation d'un nombre limité de librairies qui sont partagées par les applications développées en interne, notamment (i) les librairies d'appel de procédure à distance, (ii) les librairies de gestion des fils d'exécution et (iii) les librairies de gestion des tâches asynchrones. Le dernier point est particulièrement intéressant : *Dapper* permet en effet de sauvegarder dans le TLS – mémoire locale de fil d'exécution – l'état de chaque requête traitée par le processus courant. Cela est utile notamment lorsqu'un processus traite plusieurs requêtes à la fois et est en attente pour chaque requête d'un événement asynchrone, par exemple l'arrivée d'un paquet réseau. Si l'instrumentation permet en pratique de suivre toutes les requêtes sans aucune action de la part des développeurs des composantes de

l'application répartie, les développeurs peuvent ajouter des annotations à la trace distribuée. Cela permet à la trace de contenir des informations de débogage qui seraient autrement écrites dans les fichiers journaux de l'application. Les traces peuvent ensuite être visionnées dans une interface utilisateur dédiée pour y être analysées.

Échantillonnage L'outil *Borderpatrol* cité plus haut souffrait d'une pénalité de performance non négligeable lors de la collecte des traces. Pour faciliter la mise à l'échelle et limiter le surcoût engendré par l'instrumentation, *Dapper* propose une fonctionnalité d'échantillonnage probabiliste des requêtes qui permet de choisir un compromis entre performance et collecte de traces. Par exemple, si on décide de ne tracer que 0.1% des requêtes, le débit des requêtes n'est impacté que de 0.06%. L'article évoque aussi la volonté de travailler sur une fonctionnalité d'échantillonnage adaptatif qui permettrait de ne pas pénaliser – en matière de nombre de traces collectées – les services les moins utilisés par un échantillonnage uniforme et trop agressif. En revanche, qu'il soit adaptatif ou non, un échantillonnage probabiliste agressif risque de ne pas capturer de traces correspondant à des cas d'exécution peu fréquents. Il y a alors le risque de ne capturer en majorité que des traces similaires et qui ne correspondent pas forcément à des exécutions problématiques. Un travail récent de Las-Casas et al. (2018) a étudié la possibilité de biaiser l'échantillonnage afin d'augmenter la diversité des traces collectées, en augmentant artificiellement la probabilité de traçage d'exécutions peu communes. L'approche utilise un algorithme en ligne pour ranger les différents cas d'exécution dans un arbre au fur et à mesure de leur arrivée, et nécessite d'avoir une mesure de distance pour évaluer la similarité de deux traces données. Rien de tel n'est implémenté dans les traceurs distribués actuels mais ce travail préliminaire est encourageant et souhaitable.

Limitations L'article qui présente *Dapper* s'attarde de manière pertinente sur quelques limitations de l'outil. Une des limitations en particulier motive le travail présenté dans ce mémoire. Au moment de la publication de l'article, *Dapper* ne permettait pas de récolter des événements au niveau du noyau et de les analyser conjointement avec les traces distribuées, ce qui permettrait d'améliorer la recherche de la cause racine d'une latence.

L'initiative *OpenTracing* Plusieurs outils se sont plus ou moins directement inspirés de *Dapper* pour proposer des solutions de traçage distribué librement utilisables. *Zipkin* (Zipkin, n.d.) a été rendu disponible au public par Twitter en 2012, et *Jaeger* (Jaeger, n.d.) par Uber en 2017. L'initiative *OpenTracing* (OpenTracing, n.d.) a également vu le jour et propose (i) des interfaces de programmation dans différents

langages pour faciliter l'instrumentation des applications réparties et (ii) la spécification d'un modèle de données et de propagation de contexte. Les applications instrumentées avec *OpenTracing* peuvent l'être indépendamment du traceur utilisé, puisque n'importe quel traceur compatible avec l'API de *OpenTracing* pourra utiliser l'instrumentation effectuée. Par le biais de son traceur *Jaeger* notamment, Uber est un contributeur majeur de *OpenTracing* et utilise le traçage distribué de manière intensive pour détecter la cause des latences observées en production (Shkuro, 2017). Pour rappel, l'interface de visualisation de trace de *Jaeger* est montrée dans la figure 4.1. *Jaeger* implémente les mêmes principes que *Dapper* tant au niveau de l'architecture de collecte des traces – voir figure 2.1 – qu'au niveau de la propagation de contexte dans les requêtes – voir figure 2.2.

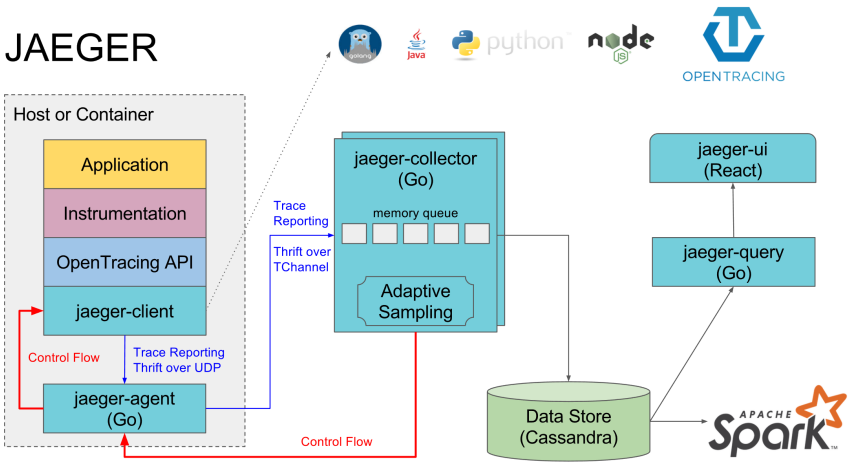

Figure 2.1 **Schéma de l'architecture du traceur distribué *Jaeger*.** Le traceur *Jaeger* repose sur un client de traçage situé dans l'application cible ainsi qu'un agent situé sur le même hôte. Les événements sont mis en lots et transmis au collecteur pour être sauvegardés dans une base de données. De la même manière que *Dapper*, *Jaeger* fait de l'échantillonnage de requêtes pour n'en tracer qu'une partie. Source : Documentation de *Jaeger* (Jaeger, 2019), distribuée sous licence Apache 2.0.

En résumé

— Le traçage distribué naît du besoin de suivre des requêtes de bout en bout dans une application répartie.

— Les premiers travaux comme *Magpie* (Barham et al., 2004) et *X-Trace* (Fonseca et al., 2007) proposent de limiter le nombre d'applications instrumentées, en se focalisant sur les intergiciels de communication, et de propager

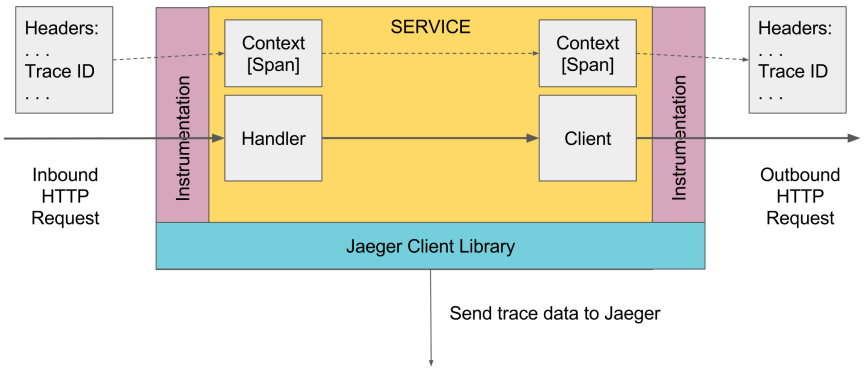

Figure 2.2 **Propagation du contexte de traçage dans les requêtes instrumentées par *Jaeger*.** Le contexte de traçage est habituellement injecté par *Jaeger* dans l'en-tête de la requête, d'où il pourra être extrait par d'autres composants de l'application répartie. Conformément à ce que préconise *Dapper*, le contexte de traçage suit exactement le même chemin que les requêtes, alors que les événements collectés suivent un chemin dédié. Source : Documentation de *Jaeger* (Jaeger, 2019), distribuée sous licence Apache 2.0.

le contexte de traçage dans les messages des requêtes elles-mêmes.

— L'outil *Dapper* de Google (Sigelman et al., 2010) est encore une référence dans le domaine et a proposé une architecture de collecte de traces centrée sur la performance et la mise à l'échelle. On trouve notamment dans le travail l'idée d'échantillonner un petit pourcentage de requêtes afin de limiter le nombre d'événements traités.

— *OpenTracing* est une initiative de spécification libre directement inspirée de *Dapper*.

2.1.3 Applications et variations autour du traçage distribué

Le but de cette section est de présenter les travaux qui ont pour but de tirer profit de la collection de traces distribuées. Nous considérons qu'il y a collection de traces distribuées dès lors que les traces permettent de reconstruire séparément chaque requête utilisateur traitée par le système. Nous verrons que pour parvenir à collecter des traces distribuées, certains outils dérogent aux principes posés par *Dapper* afin de mieux s'adapter aux spécificités du système ciblé.

Profilage de transactions Un des buts du traçage distribué est de pouvoir isoler chaque transaction traitée, souvent dans le but d'analyser un cas d'exécution particulier pro-

blématique et potentiellement rarement observé. L'approche inverse consisterait à s'intéresser non pas à des cas particuliers précis mais à une vue d'ensemble de l'application répartie. On cherche alors à obtenir une vue synthétique de la performance de l'application répartie, en moyenne, afin de connaître les endroits les plus lents de l'application que l'on cherchera à optimiser en priorité. L'outil *Whodunit* de Chanda, Cox et Zwaenepoel (2007) a précisément pour but de faire du profilage de transaction. Le travail est antérieur à *Dapper* et utilise donc des techniques différentes pour collecter les traces distribuées. *Whodunit* met en effet à profit l'instrumentation de librairies de synchronisation et de communication entre processus pour reconstruire les transactions en attribuant à chacune un identifiant unique. Le vrai apport du travail par rapport à *Dapper* réside dans la génération, pour chaque cas d'utilisation d'une application distribuée, d'un graphe de profilage qui résume le temps passé dans chaque composante de l'application. Cela permet aux développeurs de voir rapidement quelle composante de l'application est la plus lente, ou encore d'évaluer l'impact concret d'une mise à jour sur la performance de l'application.

Gestion des ressources sur une machine De la même manière, *Retro* (Mace, Bodik et al., 2015) utilise les informations de trace dans le seul but de fournir à l'utilisateur une information synthétique plutôt que la possibilité d'analyser chaque requête séparément. Dans le cas de *Retro*, l'utilisateur a accès a une mesure de contention et de charge pour des ressources telles que le CPU, la mémoire, le réseau et le disque. L'utilisateur a la possibilité de définir des flux opérationnels, qui sont des abstractions regroupant des charges de travail considérées comme similaires – parce qu'elles proviennent d'un même utilisateur, ou parce qu'elles correspondent à une même tâche. Les traces correspondant à un même flux opérationnel sont agrégées pour que l'utilisateur puisse avoir accès aux mesures de contention et de charge par flux. Enfin, *Retro* fournit un mécanisme de rétroaction – d'où le nom de l'outil – qui permet à l'utilisateur de limiter la consommation des ressources partagées par flux opérationnel. Cela permet de prioriser une charge de travail d'un certain type sur une machine qui est autrement partagée par plusieurs services, ce qui est une situation courante dans des déploiements dans le nuage. *Retro* fournit à cet effet plusieurs politiques d'ordonnancement des tâches qui sont appliquées par des points de contrôle déployés sur chaque noeud. Les auteurs montrent l'intérêt de *Retro* pour limiter la contention générée par les tâches les plus gourmandes d'une grappe de calcul distribué *Hadoop*. L'outil a par ailleurs un impact limité sur la performance, avec un surcoût mesuré – en matière de latence et de bande passante – à moins de 2%.

Analyse de chemin critique dans des systèmes hétérogènes Le travail de Chow et al. (2014) sur l'outil *Mystery Machine* de Facebook met en lumière une des limitations de *Dapper*, qui s'appuie sur l'instrumentation d'un petit nombre de librairies qui sont censées être partagées par toutes les composantes de l'application distribuée. C'est le cas chez Google, qui présente une infrastructure distribuée homogène s'appuyant sur des intergiciels de communication standardisés en interne, mais pas chez Facebook qui privilégie des systèmes hétérogènes et évoluant rapidement. Le défi que résolvent Chow et al. (2014) est donc l'obtention de traces distribuées sans instrumenter pour autant les différentes composantes de l'application, qui changent trop rapidement et sont trop différentes les unes des autres. *Mystery Machine* prend le parti de reconstruire les requêtes à partir des seuls fichiers journaux produits par l'application. Une première étape consiste à reconstruire un modèle causal que toutes les requêtes de l'application sont censées respecter, et ceci à l'aide des opérateurs (i) *"happens-before"* (se déroule avant) (ii) *"mutual exclusion"* (exclusion mutuelle) et (iii) *"pipeline"* (exécution en pipeline). Le modèle est calculé à partir des fichiers journaux de l'application est et considéré valide jusqu'à la découverte d'un contre-exemple, ce qui peut arriver lorsque l'application est mise à jour et que le modèle doit effectivement être changé en conséquence. Une fois le modèle obtenu, les fichiers journaux permettent de calculer les résultats de plusieurs analyses particulièrement utiles.

- L'analyse de chemin critique de l'application est effectuée sur un grand nombre de traces puis agrégée afin d'aider les développeurs à améliorer la latence globale de l'application.
- L'analyse des ressources sous-utilisées (en anglais, *"slack analysis"*) calcule pour chaque sous-requête la durée de laquelle on pourrait l'allonger sans que cela affecte la longueur totale du chemin critique. Cette analyse est utile pour mettre en lumière des composantes de l'application répartie qui sont soit sous-utilisées, soit optimisées prématurément. L'analyse est généralement agrégée sur un grand nombre de traces pour en présenter une vue synthétique.
- La détection d'anomalie permet de détecter les traces qui présentent une déviation importante par rapport au modèle ou au chemin critique moyen observé sur un grand nombre de traces. Elle est utilisée pour analyser et optimiser un petit pourcentage de requêtes dont la latence est inhabituellement grande.

L'approche est particulièrement flexible et est très adaptée à des architectures fortement hétérogènes. Nous nous intéressons dans notre travail à des systèmes homogènes

dont on peut instrumenter les intergiciels, mais les analyses présentées par Chow et al. (2014) sont particulièrement intéressantes et restent applicables dans le cas qui nous intéresse.

Exécution d'analyses arbitraires dans des systèmes hétérogènes Quelques années après Chow et al. (2014), Facebook présente l'évaluation de son outil *Canopy* (Kaldor et al., 2017) qui propose une approche légèrement différente. Les contraintes restent similaires, et l'outil doit notamment s'adapter à la forte hétérogénéité des applications réparties chez Facebook. Contrairement à *Mystery Machine*, qui répondait au défi en analysant des fichiers journaux collectés de manière passive, *Canopy* nécessite l'action des développeurs pour instrumenter les différentes composantes de l'application. Pour s'adapter à l'hétérogénéité des composantes – en matière de langages de programmation, de sémantique de communication, d'intergiciels utilisés –, l'outil propose différentes interfaces de programmation aux développeurs et tolère le fait que certaines composantes soient instrumentées de manière moins précise que les autres. De la même manière que *Dapper* (Sigelman et al., 2010) et *OpenTracing* (OpenTracing, n.d.), *Canopy* échantillonne les traces et nécessite que chaque transaction soit distinguée par un identifiant unique, qui est propagé par les développeurs entre les sous-requêtes d'une même transaction. Mais alors que Sigelman et al. (2010) se concentrent sur la collecte et la sauvegarde des traces, *Canopy* propose une architecture nouvelle dont le but est l'analyse de chacune des traces immédiatement après leur collecte. Chaque événement est envoyé à un serveur d'analyse, choisi en fonction de l'identifiant de trace associé à la transaction. Une fois que l'entièreté de la trace a été reçue par le serveur, ce dernier exécute une série d'analyses définies par les utilisateurs de *Canopy* et dont le rôle est d'extraire les caractéristiques de la trace collectée. Ces caractéristiques sont ensuite sauvegardées pour pouvoir être consultées plus tard dans des tableaux de bord dédiés. Cette fonctionnalité de *Canopy* est intéressante, car elle permet de synthétiser l'information issue des traces, en utilisant des fonctions et des analyses définies par l'utilisateur, et qui peuvent être modifiées dynamiquement au gré de besoins. L'approche se situe à mi-chemin entre le traçage distribué et le monitorage ; c'est particulièrement adapté pour une entreprise de la taille de Facebook pour laquelle les ingénieurs ne peuvent pas parcourir manuellement toutes les traces collectées. Une information de synthèse rapidement accessible permet ensuite d'approfondir l'analyse à l'aide des traces sauvegardées lorsqu'une anomalie est détectée. On notera aussi que *Canopy* propose des politiques d'échantillonnage plus fines que l'échantillonnage probabiliste d'*OpenTracing* (OpenTracing, n.d.). L'article parle en particulier d'"échantillonnage

opportuniste" pour qualifier une décision d'échantillonnage qui n'est pas prise lors de la création de la requête, mais plus tard lorsqu'on dispose de suffisamment d'informations pour appliquer la politique d'échantillonnage. Les événements sont collectés par défaut tant que la décision n'est pas prise, puis sont persistés ou au contraire abandonnés au besoin. Kaldor et al. (2017) notent toutefois que l'approche n'est utilisée que sur certains systèmes qui reçoivent relativement peu de requêtes et qui ne sont donc pas trop impactés par la collecte initiale de tous les événements de trace.

Exécution de requêtes arbitraires par instrumentation dynamique Un dernier travail pertinent est celui sur l'outil *Pivot Tracing* de Mace, Roelke et Fonseca (2018). Il partage des similitudes avec les deux travaux précédents en cela qu'il (i) définit un opérateur de causalité *"happened-before join"* comparable à l'opérateur *"happens-before"* de Chow et al. (2014) et (ii) permet à l'utilisateur de définir arbitrairement de nouvelles métriques et analyses sans interruption du service. L'utilisateur est en effet en mesure de formuler des requêtes qui combinent les mesures de différentes variables présentes en n'importe quel endroit de l'application distribuée. La requête peut faire intervenir l'opérateur *"happened-before join"* qui permet de filtrer les transactions que l'on veut prendre en compte dans le calcul de la métrique. Pour que la métrique puisse être calculée, les variables d'intérêt sont échantillonnées en ajoutant dynamiquement du code d'instrumentation aux processus de l'application en cours d'exécution. La métrique nouvellement calculée par la requête est ensuite envoyée sous forme de flux à une interface utilisateur, qui permet de visualiser l'évolution de la métrique dans le temps à partir du moment de sa création. La méthode est puissante, car elle permet de calculer des métriques arbitraires complexes en insérant des points de trace de manière dynamique dans le code de l'application. Cependant, l'implémentation proposée par Mace, Roelke et Fonseca (2018) est limitée aux applications écrites en Java, car elle met à profit les capacités de remplacement de code "à chaud" de la machine virtuelle Java.

En résumé

— Le traçage distribué peut servir de base à des outils qui analysent la consommation des ressources par une application. Il peut s'agir d'outils comme *Whodunit* (Chanda, Cox et Zwaenepoel, 2007), qui calcule le profil de performance d'une application, ou *Retro* (Mace, Bodik et al., 2015) qui permet d'imposer dynamiquement des limites de ressources à une application.

— Facebook est à l'origine de plusieurs travaux visant à adapter le traçage distribué

à des systèmes fortement hétérogènes, dont *Mystery Machine* (Chow et al., 2014) et *Canopy* (Kaldor et al., 2017).

— Quelques outils proposent à l'utilisateur de définir des analyses arbitraires qui sont exécutées au moment de la collecte des événements ou juste après la collecte de la trace. C'est le cas de *Canopy* (Kaldor et al., 2017), qui nécessite d'avoir défini ces analyses au préalable, ou encore de *Pivot Tracing* (Mace, Roelke et Fonseca, 2018) qui peut insérer dynamiquement des sondes d'analyse dans l'application.

2.2 Techniques de traçage logiciel

Le travail présenté dans ce mémoire s'appuie sur la collecte et l'analyse combinée de traces distribuées et de traces d'exécution dans le noyau Linux. Le traçage distribué a été discuté dans la section précédente de la revue de littérature. La section qui suit est donc consacrée aux différentes solutions qui existent pour tracer des événements dans le noyau Linux, puis pour analyser les événements collectés. Enfin, notre travail requiert l'instrumentation d'outils dans l'espace utilisateur ; il faudra donc privilégier les traceurs qui ont la capacité de collecter des événements dans le noyau et dans l'espace utilisateur en même temps.

2.2.1 Collecte des événements de trace

Dans un premier temps, nous nous intéresserons aux outils et aux techniques utilisés pour la collecte de traces dans les systèmes Linux.

`strace` et `ltrace`

`strace` et `ltrace` sont deux outils simples pour tracer respectivement les appels système et les appels aux fonctions de librairies standards faits par une application. Ce sont des outils intéressants et principalement utilisés pour comprendre rapidement le comportement d'un programme, même sans avoir le code source disponible. Ils reposent tous les deux sur l'infrastructure de traçage `ptrace` offerte par Linux, qui permet de s'attacher à un programme pour intercepter les appels faits à une fonction cible. Le gros défaut de cette technique est qu'elle génère une interruption à chaque fois que la fonction en question est exécutée, de manière à pouvoir intercepter l'appel et, dans le cas de `strace` et `ltrace`, noter l'événement correspondant. Gregg (2014) note que le mécanisme `ptrace`, sur lequel repose par ailleurs le débogueur `GDB`, induit un surcoût tel que les outils qui reposent sur ce mécanisme sont en pratique inutilisables sur des systèmes en production. Selon Gebai et M. R. Dagenais (2018), un appel système `ioctl` peut être 19 fois plus lent lorsqu'il est tracé avec `strace`. L'impact est trop important pour utiliser ces deux outils en production sur des applications distribuées.

`ftrace`

L'outil `ftrace` est présent par défaut dans les systèmes Linux et permet de faire du traçage d'événements dans le noyau. Il s'appuie sur des points de trace statiques insérés dans le noyau Linux par les développeurs, et permet également à l'utilisateur de fournir ses propres points de trace grâce à des modules noyau chargés dynamiquement, les `kprobes`. Il est à noter

que les points de trace statiques insérés dans le noyau le sont sous la forme d'une macro `TRACE_EVENT` (Rostedt, 2017) qui est partagée avec d'autres traceurs noyau. Comme pour les groupes de contrôle qu'utilisent les conteneurs Linux et qui sont pilotables depuis le système de fichiers `cgroup` exposé par le noyau, les points de trace `ftrace` peuvent être contrôlés depuis le système de fichiers `tracefs` (Aranya, Wright et Zadok, 2004). L'outil est donc particulièrement adapté pour les administrateurs de systèmes qui veulent diagnostiquer une anomalie en lisant les fichiers de trace disponibles par le système de fichiers `tracefs`. En revanche, il ne permet pas de faire de traçage dans l'espace utilisateur et ne pourrait donc pas être utilisé seul dans le cadre de notre travail. Gebai et M. R. Dagenais (2018) notent par ailleurs que si la latence du traceur est particulièrement bonne lorsqu'un point de trace est atteint, la performance se dégrade sur des systèmes multicœurs qui sont fréquents dans les infrastructures infonuagiques.

perf

De la même manière que `ftrace`, `perf` est un outil directement intégré dans le noyau Linux. Il peut utiliser les points de trace statiques déjà présents dans le noyau – comme `ftrace` le fait –, mais son intérêt principal réside dans l'échantillonnage de compteurs de performance matériels d'un processus donné. `perf` peut par exemple enregistrer l'évolution du nombre de fautes de caches ou de TLB, ce qui est très utile pour mesurer à quel point une application est optimisée pour le matériel qui l'exécute. Ghods (2016) note par ailleurs que `perf` est maintenant un outil complet qui intègre plusieurs analyses permettant de faire du profilage ou encore de détecter des contentions dues à des verrous. L'outil permet d'insérer des points de trace dynamiquement dans le noyau – avec le mécanisme `kprobes` – ainsi que dans des applications dans l'espace utilisateur – avec `uprobes`. Il faut toutefois noter que `uprobes` peut avoir un impact notable sur la performance d'une application.

eBPF

Un outil qu'il est intéressant de mentionner même si ce n'est pas un traceur à proprement parler est `eBPF` (Fleming, 2017). De la même manière que `BPF` (McCanne et Jacobson, 1993), qui est utilisé pour filtrer des paquets réseau de manière efficace, `eBPF` permet à l'utilisateur d'écrire des portions de code qui sont compilées en langage intermédiaire puis insérées à des endroits arbitraires du noyau. Le code écrit par l'utilisateur a accès aux structures de données internes au noyau et peut effectuer des calculs simples dont les résultats peuvent être surveillés par l'utilisateur. Gebai et M. R. Dagenais (2018) montrent que le surcoût à l'exécution est comparable à celui de `perf` tout en étant particulièrement flexible puisqu'il

permet l'exécution de code arbitraire dans le noyau à des fins de monitorage. Si nous n'avons pas utilisé `eBPF` dans notre travail, il est tout à fait pertinent d'envisager son utilité à des fins d'analyse pour insérer des sondes dynamiquement dans le noyau, sans perturber pour autant sur la durée les systèmes ciblés en production. Sharma et M. Dagenais (2016) ont également étudié la validité de l'approche pour activer ou non les points de trace selon des règles arbitrairement définies – et évaluées dynamiquement –, afin de réduire la quantité d'événements collectés par les traceurs usuels aux seuls événements intéressants.

SystemTap

`SystemTap` (Prasad et al., 2005 ; Eigler et Hat, 2006) partage beaucoup de points communs avec `eBPF`. Il permet notamment à l'utilisateur d'écrire du code arbitraire qui est ensuite chargé dynamiquement dans le noyau pour y servir de sonde. De la même manière que `eBPF`, `SystemTap` est particulièrement flexible et adapté à du prototypage et à du monitorage rapide, car il ne nécessite pas de recompilation du noyau, comme c'est le cas lorsque des points de trace statiques sont modifiés. L'outil permet de plus de faire du traçage d'applications dans l'espace utilisateur de la même manière, en insérant dynamiquement du code arbitraire dans le programme cible, grâce à l'infrastructure `uprobes` du noyau Linux. Cependant, et de la même manière que pour `ftrace` et `eBPF`, Gebai et M. R. Dagenais (2018) notent que la performance de `SystemTap` est significativement impactée dans le cas d'un système qui a un grand nombre de cœurs.

LTTng

Depuis qu'il a succédé à `LTT`, `LTTng` (Desnoyers et M. R. Dagenais, 2006) est devenu un traceur de référence pour Linux. Contrairement à la plupart des outils évoqués ci-dessus, `LTTng` s'appuie sur des points de trace statiques, dans le noyau comme dans les applications de l'espace utilisateur instrumentées. Il peut également collecter les événements correspondant à des points de trace dynamiques fournis avec le mécanisme `kprobes`. Un aspect important de `LTTng` est qu'il repose sur le travail des développeurs pour l'instrumentation manuelle des applications dans l'espace utilisateur, et qu'il requiert que les applications soient recompilées pour pouvoir prendre en compte les points de trace. L'impact pour nous sera limité, car il s'agira d'instrumenter un petit nombre d'intergiciels que l'on suppose utilisés dans un système homogène. Dans un système fortement hétérogène, qui ne privilégie pas la standardisation, `LTTng` ne serait sans doute pas la solution la plus adaptée. En ce qui nous concerne, `LTTng` présente un nombre non négligeable d'avantages qui en font un choix sûr pour notre travail.

Format de trace standardisé et performant `LTTng` s'appuie sur le *Common Trace Format* (DiaMon, n.d.[b]), un format de trace standardisé qui est implémenté par l'outil `Babeltrace` (DiaMon, n.d.[a]). Le format est conçu pour qu'il soit possible d'écrire très rapidement dans les fichiers de trace et ainsi de supporter des débits d'événements élevés. C'est un format binaire qui permet donc de limiter la taille des traces sur le disque, mais les champs des événements de trace peuvent être personnalisés à l'aide d'un langage descriptif proche du C. L'avantage d'un format standardisé est qu'il sépare clairement la phase de collecte de la trace de la phase d'analyse, et qu'il est donc possible de (i) collecter des traces sans se préoccuper du logiciel utilisé pour l'analyse et (ii) développer des analyses des traces indépendantes du traceur qui les a collectées.

Architecture centrée sur la performance Contrairement à des outils comme `eBPF` et `SystemTap` qui sont orientés vers le monitorage et le calcul en temps réel de métriques pertinentes, l'objectif de `LTTng` est la collecte d'événements de très bas niveau de manière précise et exhaustive pour pouvoir en faire une analyse hors-ligne. Pour pouvoir reconstruire *a posteriori* les métriques que d'autres outils collectent en temps réel, le traceur doit donc obéir aux contraintes qui suivent.

— Il collecte tous les événements pertinents pour l'analyse de trace. Par exemple, pour être en mesure de calculer le temps CPU utilisé par un processus, il faut récolter tous les événements `sched_switch` correspondant à un changement de contexte depuis ou vers ce processus.

— Il collecte les événements avec une précision qui permet les analyses. Par exemple, une précision d'une microseconde n'est pas envisageable pour tracer des événements noyau, sachant qu'un appel système Linux peut être exécuté en aussi peu de temps qu'une centaine de nanosecondes.

— Il est capable de gérer un débit d'événements très important et d'autant plus élevé que la charge sur le système est élevée.

— Il doit avoir un surcoût de performance raisonnable et surtout ne pas perturber le comportement du système par l'observation de ce système. En particulier, certains problèmes reposent sur un ordonnancement des événements tellement fin qu'ils peuvent disparaître si un observateur modifie cet ordonnancement par sa simple présence.

D'après Desnoyers et M. R. Dagenais (2006), `LTTng` répond à ces problématiques et a en particulier un surcoût d'environ 2% pour des charges de travail élevées. Il utilise des estampilles de temps précises à la nanoseconde, et collecte les événements à la fois

à partir des points de trace statiques du noyau, et à partir de points de trace ajoutés au besoin manuellement par les développeurs des applications. D'après un comparatif de performance des traceurs récemment publié par Gebai et M. R. Dagenais (2018), `LTTng` est le traceur noyau le plus performant pour Linux. Un choix fait par les développeurs de `LTTng` est de privilégier la performance du système – et donc le fait de ne pas perturber son comportement –, quitte à perdre des événements de trace si le débit d'événements est trop élevé pour pouvoir persister à temps les événements sur disque (Desnoyers et M. R. Dagenais, 2012).

Mise à l'échelle sur les systèmes multicœurs Gebai et M. R. Dagenais (2018) notent que quelques traceurs, parmi lesquels `SystemTap`, `ftrace` et `eBPF`, voient leur performance se dégrader lorsque le nombre de cœurs augmente sur le système. À l'inverse, `LTTng` est conçu pour se mettre à l'échelle sans difficulté sur un grand nombre de cœurs : des tampons circulaires propres à chaque cœur de processeur sont utilisés pour sauvegarder temporairement les événements – comme illustré dans la figure 2.3 –, et les opérations de synchronisation sont minimisées pour limiter la communication entre les cœurs (Desnoyers et M. R. Dagenais, 2012).

Paramétrage des sessions de traçage En plus de proposer aux développeurs une librairie – disponible en Java, en C, en C++ et en Python – pour instrumenter leurs applications dans l'espace utilisateur, `LTTng` est particulièrement configurable lorsqu'il s'agit de tracer un système. L'utilisateur peut définir un nombre arbitraire de sessions de traçage, chacune permettant à l'utilisateur de n'activer que quelques points de trace bien précis. Il peut également décider d'attacher à chaque événement de trace un contexte – PID ou TID du processus concerné, nombre de défauts de cache L1, etc. – qui sera enregistré avec l'événement dans la trace CTF. Il est possible de régler la taille et le nombre des tampons circulaires utilisés pour sauvegarder les événements de trace, en prévision par exemple d'un volume important d'événements.

Prise d'instantanés `LTTng` offre une fonctionnalité qui permet de limiter l'impact du traçage sur l'activité du disque dur sans pour autant sacrifier la collecte d'événements. Alors que les événements présents dans les tampons circulaires sont usuellement vidés régulièrement pour être copiés dans les fichiers de traces CTF, le mode de prise d'instantané – en anglais "snapshot mode" – ne copie rien par défaut sur le disque. Les événements sont collectés en continu et copiés dans les tampons circulaires – en écrasant possiblement les événements les plus anciens si le tampon est plein – ; si l'uti-

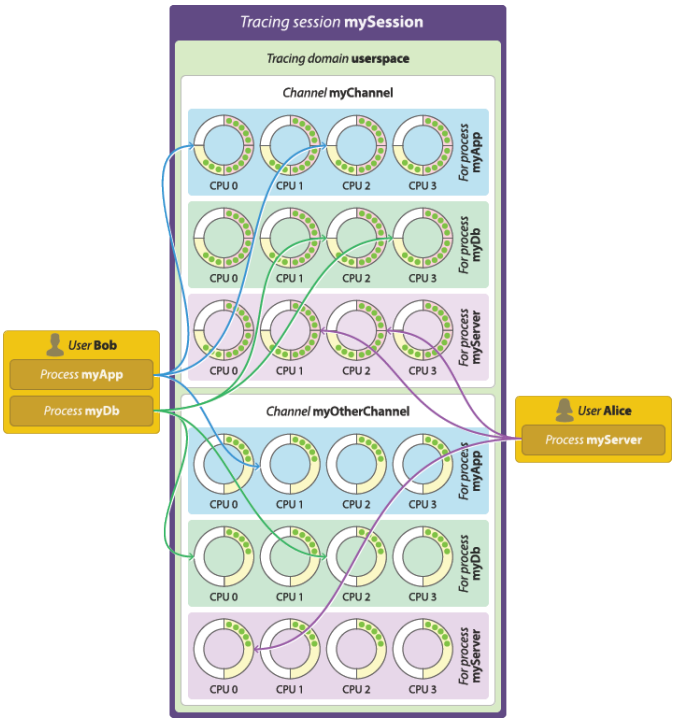

Figure 2.3 **Schéma de la configuration des tampons d'événements du traceur *LTTng*.** Les tampons d'événements du traceur *LTTng* sont propres à chaque cœur de processeur pour une meilleure performance. Ils sont de plus spécifiques, au choix, soit à chaque utilisateur, soit à chaque processus comme c'est le cas ici. Source : Documentation de *LTTng* (LTTng, 2018), distribuée sous licence CC-BY 4.0.

lisateur demande un instantané, alors à ce moment seulement le contenu des tampons sera copié dans un fichier de trace. En réglant correctement la taille des tampons, il est possible d'obtenir une trace de quelques secondes sur demande, usuellement lorsque l'utilisateur détecte une anomalie sur le système. Desfossez, Desnoyers et M. R. Dagenais (2016) ont utilisé cette fonctionnalité de `LTTng` pour détecter et permettre l'analyse de latences. Le traceur est configuré en mode de prise d'instantané, et l'utilisateur peut définir des conditions arbitraires pour enregistrer une trace au besoin et pouvoir l'analyser plus tard. Desfossez, Desnoyers et M. R. Dagenais (2016) ont par exemple créé un module noyau qui surveille la durée des appels système, et qui déclenche une prise d'instantané `LTTng` si un des appels dépasse une durée seuil. C'est une approche particulièrement intéressante qui peut être appliquée conjointement avec du traçage distribué : de la même manière, si une requête utilisateur dépasse une durée seuil, alors on déclenche une prise d'instantané à des fins d'investigation.

Développement actif `LTTng` est un outil libre qui est activement maintenu par l'entre-

prise EfficiOS (LTTng, n.d.). En particulier, certaines fonctionnalités intéressantes pour les systèmes répartis sont en cours de développement et devraient voir le jour dans une des prochaines versions du traceur. Un meilleur support des conteneurs Linux (Jeanson, 2019) permettra au travail présenté dans ce mémoire d'être étendu à des applications réparties déployées à l'aide d'outils populaires comme *Docker* ou *Kubernetes* (Bernstein, 2014).

Évolution des techniques d'instrumentation dynamique

Un domaine de recherche en évolution est l'étude des techniques d'instrumentation dynamique d'applications. Depuis les premiers travaux de Cantrill, Shapiro, Leventhal et al. (2004) sur `DTrace`, la plupart des outils et infrastructures d'instrumentation dynamique – notamment `uprobes` sur lequel reposent `perf` et `SystemTap` – ont suivi le même mécanisme de collecte de trace. Une fois atteint lors de l'exécution, le point de trace inséré dynamiquement va provoquer une interruption logicielle – et donc un passage en mode noyau – dont la gestion est définie par le traceur. Le désavantage majeur de cette technique est son impact sur la performance, puisqu'il faut régulièrement basculer en mode noyau puis en mode utilisateur à chaque fois qu'une instruction instrumentée est exécutée.

Plus récemment, des équipes de recherche ont étudié des moyens moins coûteux de collecter des traces utilisateur à l'aide de points de trace insérés dynamiquement. L'intérêt est particulièrement grand pour les systèmes en production qui ont des contraintes de disponibilité. Si les techniques ne sont pas encore assez développées pour qu'on envisage leur utilisation dans le cadre de notre travail, l'approche de Chamith et al. (2017) semble prometteuse, car elle fonctionne sans interruption logicielle. Les lignes d'assembleur visées par l'instrumentation sont remplacées par des instructions de saut qui redirigeront l'exécution vers un emplacement bien choisi du code, où l'outil aura inséré le code d'instrumentation. L'intérêt du travail réside dans les techniques conçues pour contourner les limites de l'architecture `x86-64` en ce qui concerne les instructions de saut. Chamith et al. (2017) proposent d'utiliser des trampolines successifs pour effectuer des sauts plus loin que l'architecture ne le permet et ainsi pouvoir atteindre le code d'instrumentation dans tous les cas. De plus, le surcoût mesuré à l'exécution est inférieur à 1%.

En résumé

— Les traceurs comme `strace` et `ltrace`, qui reposent sur le mécanisme `ptrace` du noyau Linux, ont un impact trop grand sur la performance pour qu'il soit

envisageable de les utiliser en production.

— `perf` est un traceur puissant et complet qui est notamment capable d'enregistrer la valeur de compteurs de performance pertinents. En revanche, son instrumentation d'applications dans l'espace utilisateur repose sur le mécanisme `uprobes`, qui peut causer des ralentissements.

— Des outils comme `eBPF` et `SystemTap` sont particulièrement utiles, car ils permettent d'insérer du code d'instrumentation arbitraire dans le noyau et les applications. Ils sont en revanche moins orientés vers la collecte d'événements insérés statiquement et conviendront moins à notre utilisation.

— `LTTng` est un traceur complet pour Linux, orienté vers la performance. Ses nombreuses fonctionnalités de paramétrage des sessions de traçage ainsi que sa performance sur les systèmes multicœurs en font l'outil le mieux adapté à nos besoins.

— Le développement de solutions d'instrumentation dynamique pour l'espace utilisateur est encourageant, notamment pour éviter le surcoût de performance causé par `uprobes`.

2.2.2 Analyse des traces

Le rôle des outils que nous avons étudiés ci-dessus est la simple collecte d'événements non traités et de bas niveau. Selon la charge de travail du système ciblé et le nombre de points de trace activés, un traceur tel que `LTTng` peut produire des traces de plusieurs centaines de mégaoctets de données pour quelques secondes d'exécution seulement, que l'on ne peut pas envisager d'analyser manuellement. Le rôle des outils d'analyse de trace est justement de parcourir les traces selon des algorithmes définis pour en proposer une synthèse ou une vue plus utile à l'utilisateur.

Un bon exemple d'algorithme d'analyse de trace est le calcul – à partir des seuls événements de trace noyau – de métriques de plus haut niveau, par exemple le taux d'utilisation CPU. Ces métriques peuvent usuellement être accédées par des outils de monitorage, mais la fréquence d'échantillonnage est fixée lors de l'exécution et il n'est pas possible de rendre les métriques plus précises à des fins d'analyse. Giraldeau, Desfossez et al. (2011) proposent de calculer ces métriques, avec une précision arbitraire choisie lors de l'analyse, à partir des traces noyau collectées sur le système ciblé. Dans le cas de l'utilisation CPU, il suffit de ne considérer que les événements de la trace qui correspondent à un changement de contexte. Ces événements contiennent l'identifiant du processus qui devient actif, ainsi que l'identifiant du cœur de

CPU sur lequel il est ordonnancé. Il est ainsi possible, pour chaque processeur, de calculer tous les intervalles qui correspondent à l'exécution d'un processus, et ensuite à partir de là le taux d'utilisation CPU sur un intervalle donné. Des outils d'analyse de trace modernes comme *Trace Compass* (Eclipse, n.d.) incluent ce type d'analyse pour des traces noyau, et en proposent une vue interactive que l'utilisateur peut parcourir pour cerner les anomalies présentes lors de l'exécution. Il est par exemple possible de cibler d'un coup d'œil le moment de la trace qui correspond à une allocation totale de mémoire particulièrement élevée pour en trouver la cause racine. Certaines analyses permettent également de détailler les changements d'état d'un fil d'exécution au cours du temps, comme montré dans la figure 2.4.

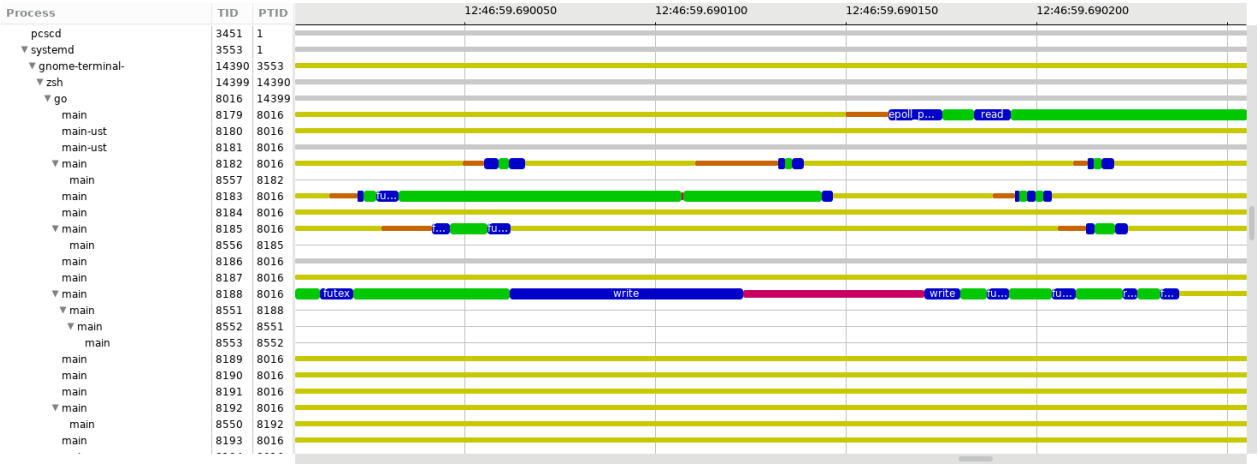

Figure 2.4 **Vue de l'évolution de l'état des fils d'exécution dans l'outil d'analyse de trace *Trace Compass*.** La vue de flux de contrôle – *Control Flow View* en anglais – de *Trace Compass* permet d'observer l'évolution de l'état de chaque fil d'exécution suivi. Chaque état est représenté par une couleur différente, par exemple jaune (fil d'exécution bloqué), orange (en attente de CPU), vert (en cours d'exécution) ou encore bleu (exécution d'un appel système).

Des analyses plus complexes peuvent utiliser des machines à état pour reconstruire la suite des états d'un système à partir des événements trouvés dans la trace. Montplaisir-Gonçalves et al. (2013) proposent une structure de données efficace et maintenant utilisée dans *Trace Compass* pour sauvegarder dans un fichier les intervalles d'états d'un système calculés par une analyse. La structure de données permet d'accéder à un état en un temps logarithmique et est donc particulièrement adaptée aux requêtes successives engendrées par un utilisateur parcourant une vue dans *Trace Compass*. L'intérêt d'une structure de données efficace et qui puisse être persistée sur le disque est qu'une fois les analyses effectuées, les arbres d'états peuvent être conservés pour être parcourus ultérieurement à travers les vues de l'outil. Il n'y

a donc pas besoin de parcourir la trace ou d'effectuer l'analyse à nouveau.

Un type d'analyse particulièrement intéressant est l'analyse de chemin critique d'un processus. Cette analyse permet de reconstruire le plus long chemin d'attente pour des ressources pour un processus donné. A chaque instant où le processus n'est pas en cours d'exécution, l'algorithme détermine la cause du blocage, qui peut être entre autres (i) l'attente d'un paquet réseau, (ii) l'attente après le disque, (iii) la préemption par un autre processus plus prioritaire, ou encore (iv) l'attente après un verrou ou une barrière de synchronisation. Giraldeau et M. Dagenais (2016) ont implémenté cette analyse et l'ont même étendue au cas de plusieurs machines exécutant une application distribuée. L'implémentation de cette analyse dans *Trace Compass* est illustrée dans la figure 2.5. L'analyse utilise les interruptions et les événements associés au réseau pour relier les chemins critiques entre des machines différentes sur lesquelles les traces noyau ont été collectées séparément. Une analyse de chemin critique adaptée du travail de Giraldeau et M. Dagenais (2016) fait partie du travail présenté dans ce mémoire.

Figure 2.5 **Vue de l'analyse de chemin critique dans l'outil d'analyse de trace *Trace Compass*.** La vue de chemin critique de *Trace Compass* permet d'observer les différents états du chemin critique d'un processus donné. Chaque état est représenté par une couleur différente, selon la nature de la ressource après laquelle le processus attend. Un état bleu signifie par exemple que le processus attend après un événement d'horloge. Les flèches permettent de suivre le chemin critique lorsqu'il s'étend sur un autre processus.

Dans un autre registre, on trouve des analyses qui utilisent au mieux la collecte combinée de traces d'exécution noyau et de traces applicatives. Les traces issues de l'espace utilisateur permettent de rendre les analyses plus pertinentes en fournissant du contexte de plus haut

niveau aux événements noyau. Il est par exemple possible de compiler une application avec les compilateurs `gcc` ou `clang` de manière à ce que toutes les entrées et sorties de fonctions soient tracées. Les traces obtenues permettent d'obtenir des graphes d'appel de fonction pour chacun des processus instrumentés, et de les corréler avec d'autres événements noyau faisant partie de la trace. On peut ainsi savoir quelle fonction du programme a été bloquée suite à une demande de prise de verrou, ou encore quelle fonction a échoué un grand nombre d'appels système. Doray et M. Dagenais (2017) utilisent cette approche dans leur outil *TraceCompare* pour faire des analyses comparatives de traces d'exécution. Le but est d'identifier les différences entre deux exécutions d'une même tâche, par exemple pour comprendre l'origine d'une latence dans une des deux exécutions. L'outil peut diriger l'utilisateur vers les blocs de code qui ont un comportement différent d'une exécution à l'autre afin de faciliter la correction des anomalies. Les auteurs soulignent que l'outil est particulièrement adapté pour détecter des régressions de performance entre deux versions d'une même application. L'originalité du travail est de combiner des traces dont les événements sont de niveaux d'abstraction différents. Cela permet d'allier la précision des analyses d'événements noyau avec le contexte, nécessaire à la compréhension des analyses, fourni par des informations de plus haut niveau. Cette approche est centrale dans le travail présenté par ce mémoire.

En résumé

- Une trace noyau suffisamment précise permet de calculer *a posteriori* des métriques de performance intéressantes avec une précision arbitraire.
- Des outils tels que *Trace Compass* ont été optimisés pour pouvoir analyser de manière efficace des grandes quantités d'événements et les synthétiser sous forme d'arbre d'intervalles.
- L'analyse du chemin critique d'une application permet de faire de la recherche de cause racine de latence en précisant la raison de chaque blocage. Cette analyse peut être étendue à des applications réparties sur plusieurs machines.
- La combinaison de traces de natures différences permet de rendre les analyses plus riches et compréhensibles pour l'utilisateur.

2.2.3 Synchronisation des traces

Une problématique pour pouvoir appliquer les analyses évoquées ci-dessus à des traces qui ont été collectées soit depuis des machines différentes, soit avec des traceurs différents, est de synchroniser temporellement les traces. Poirier, Roy et M. Dagenais (2010) ont déjà utilisé

l'algorithme de l'enveloppe convexe afin de synchroniser des traces noyau collectées depuis des machines différentes. L'approche consiste à identifier des événements qui sont causalement reliés : par exemple, à un événement d'envoi d'un paquet sur le réseau peut correspondre un événement de réception sur une autre machine. Si ces événements causalement ordonnés ne le sont pas temporellement, alors on peut corriger l'horloge d'une des deux machines afin de rétablir l'ordre naturel des événements. Si les traces disposent de suffisamment d'événements causalement reliés, alors il est possible d'obtenir une correction très satisfaisante des temps des événements, ainsi qu'une estimation de la précision de l'horloge en chaque point de la trace.

Cela est plus complexe pour des traces de natures différentes. Alors qu'un traceur noyau comme `LTTng` produit des événements dont l'estampille de temps est donnée en nanosecondes, les traceurs *OpenTracing* sont souvent précis à la microseconde près. Pour positionner les événements de traces distribuées relativement à certains événements noyau critiques – comme des changements de contexte –, il faut donc des méthodes pour synchroniser explicitement les traces. Ardelean, Diwan et Erdman (2018) proposent une méthode d'injection verticale de contexte, afin d'intégrer aux traces noyau suffisamment d'informations pour pouvoir positionner les événements des traces distribuées. Ils ont modifié leur traceur distribué pour qu'il effectue un "faux" appel système à chaque événement. Leur idée est d'utiliser l'appel système `getpid`, qui ne requiert pas d'argument, et de lui passer en paramètre l'identifiant unique du contexte de l'événement. Cela donnera lieu à un événement noyau qui pourra être collecté avec une grande précision et utilisé pour synchroniser les traces. L'inconvénient majeur de la technique est qu'elle entraîne un appel système à chaque événement intercepté par le traceur distribué, et donc un surcoût qui peut être pénalisant pour un débit de requêtes élevé. Sheth, Sun et Sambasivan (2018) vont même encore plus loin en proposant de traiter tous les appels systèmes comme des requêtes engendrées par l'application tracée. Ils ont écrit un module noyau qui permet au traceur distribué *Jaeger* d'indiquer au noyau le début et la fin d'une requête. L'information est sauvegardée dans la structure `task_struct` associée au processus est pourra être lue par une version modifiée du traceur `LTTng` afin de propager l'information dans les traces noyau. Une limitation de ce travail est que la taille de la trace noyau peut rapidement augmenter si chaque événement se voit associer un identifiant de requête *Jaeger*.

En résumé

— La synchronisation de traces noyau provenant de machines physiques différentes est rendue possible en analysant les événements correspondant à des communi-

cations entre ces machines.

— La synchronisation de traces noyau avec des traces distribuées requiert une instrumentation du traceur distribué afin de propager le contexte dans la trace de plus bas niveau. Cependant, les techniques trouvées dans la littérature posent un problème de performance ou de mise à l'échelle.

Conclusion de la revue critique de la littérature

Dans cette revue critique de la littérature, nous avons étudié les approches et outils adaptés à l'analyse de la performance des applications réparties et au traçage logiciel. Nous avons pu constater d'une part que les travaux les plus précis nécessitent une instrumentation des applications ou de leurs intergiciels, et d'autre part que les applications distribuées tendent à une standardisation de leurs protocoles internes de communication. En s'appuyant sur ces constats, le traçage distribué s'est imposé, avec des outils comme *Dapper* et des initiatives comme *OpenTracing*, comme une méthode prometteuse de monitorage des latences d'une application répartie. Quelques travaux ont étudié l'utilisation du traçage distribué pour faire de l'analyse de cause racine des problèmes de latence d'une application. Ces outils sont cependant limités, car ils ne collectent pas les informations de bas niveau qui permettent de connaître la cause précise d'une latence au niveau du noyau.

Nous avons également identifié le traceur `LTTng` comme un candidat solide pour la collecte des événements au niveau du noyau. Ce traceur permettrait en plus d'instrumenter manuellement les librairies nécessaires à la synchronisation des traces distribuées et des traces noyau. Il sera, de même que *Trace Compass* pour l'analyse de trace, l'outil utilisé dans le cadre de notre travail.

CHAPITRE 3 MÉTHODOLOGIE

Dans les sections suivantes, nous détaillons l'environnement matériel et logiciel dans lequel nous avons implémenté la solution décrite dans le chapitre suivant. Nous fournissons également les liens permettant de consulter et d'utiliser le code que nous avons développé.

3.1 Environnement utilisé

3.1.1 Configuration matérielle et logicielle

Le travail décrit dans le reste de ce mémoire a été réalisé et testé sur un ordinateur de bureau Linux possédant les caractéristiques suivantes :

— système d'exploitation Fedora Workstation 29 [1], basé sur le noyau Linux 5.0 ;

— processeur central Intel® Core™ i7-7820X de fréquence 3.60 Ghz (8 cœurs physiques, 16 cœurs logiques) ;

— 32 Gio de mémoire vive DDR4 de fréquence 2400 MHz.

3.1.2 Logiciels utilisés

Nous avons utilisé le traceur *LTTng* [2] en version 2.11.0-rc1, ainsi que le traceur distribué *Jaeger* [3] – en version 0.35.5 pour le client Java, et en version 2.7.0 pour le client Go.

Les analyses de trace ont été développées dans l'outil *Trace Compass* [4] en version 5.0.0, comme fonctionnalité de l'incubateur [5].

3.2 Outils développés

3.2.1 Collecte des traces

Notre solution de collecte des traces nécessite l'instrumentation du traceur distribué *Jaeger* par le traceur en espace utilisateur *LTTng*. Nous avons implémenté notre solution sur les versions Java et Go de *Jaeger*.

1. `https://getfedora.org/fr/workstation/`
2. `https://lttng.org/`
3. `https://www.jaegertracing.io/`
4. `https://www.eclipse.org/tracecompass/`
5. `https://wiki.eclipse.org/Trace_Compass/Incubator_Guidelines`

LTTng propose une librairie d'instrumentation pour Java [6], que nous avons utilisée afin d'ajouter des points de trace au client Java pour *Jaeger*. *Notre version modifiée de ce dernier est disponible à l'adresse suivante :*

`https://github.com/loicgelle/jaeger-client-java`

Nous avons pu utiliser la version modifiée du traceur pour instrumenter l'application Cassandra. Cette dernière permet à l'utilisateur de fournir une librairie de traçage dont les fonctions seront appelées à chaque début ou fin de sous-requête. *Notre librairie de traçage est disponible à l'adresse suivante :*

`https://github.com/loicgelle/cassandra-jaeger-tracing`

LTTng ne proposant pas de librairie en Go, nous avons créé une librairie écrite en C et qui définit les points de trace que nous voulions utiliser. *Le code permettant de compiler cette librairie est disponible à l'adresse suivante :*

`https://github.com/loicgelle/jaeger-go-lttng-instr`

Nous avons ensuite pu utiliser cette librairie pour instrumenter le traceur *Jaeger*, grâce au mécanisme `cgo` [7] qui permet d'appeler des fonctions écrites en C depuis du code écrit en Go. *Notre version modifiée du traceur est disponible à l'adresse suivante :*

`https://github.com/loicgelle/jaeger-client-go`

3.2.2 Analyse dans Trace Compass

Nous avons implémenté les algorithmes de synchronisation et d'analyse de trace décrits plus bas dans l'incubateur de *Trace Compass*. *Notre version modifiée de l'incubateur est disponible à l'adresse suivante :*

`https://github.com/loicgelle/tracecompass-incubator`

6. `https://lttng.org/docs/v2.10/#doc-java-application`
7. `https://golang.org/cmd/cgo/`

CHAPITRE 4 ARTICLE : COMBINING DISTRIBUTED AND KERNEL TRACING FOR PERFORMANCE ANALYSIS OF CLOUD APPLICATIONS

Authors

Loïc Gelle <loic.gelle@polymtl.ca>
Michel R. Dagenais <michel.dagenais@polymtl.ca>
Department of Computer and Software Engineering, École Polytechnique de Montréal

Submitted to: IEEE Transactions on Parallel and Distributed Systems

Keywords: performance analysis, distributed tracing, software tracing, distributed applications

Abstract

Distributed tracing allows for tracking user requests than span across multiple services and machines in a distributed application. However, typical cloud applications rely on abstraction layers that can hide the root cause of latency, happening between processes or into the kernel. Because of its focus on high-level events, distributed tracing can be limited when trying to detect complex contentions and relate them back to the originating requests. Cross-level analyses that include kernel events are necessary to debug problems as prevalent as mutex or disk contention.

This paper describes a new solution for combining distributed tracing with low-level software tracing in order to achieve better latency root cause finding. We explain how we achieve hybrid trace collection to capture and synchronize both kernel and distributed requests events. Then, we present our design and implementation for a request critical path analysis. We show that our analysis describes precisely how each request spends its time and what stands in their critical path.

4.1 Introduction

Distributed applications have become an obvious choice for deploying services in the cloud. They offer better performance, flexibility and resilience than their monolithic counterparts.

However, they often rely on increasingly numerous abstraction and software layers like communication and RPC (Remote Procedure Call) middleware, Linux containers, virtual machines and different threading and garbage collection mechanisms offered by programming languages. All of this makes development, deployment, scaling and maintenance easier, while making debugging more complex at the same time. Tracking user requests that span multiple services on different physical machines is also a challenge.

Distributed tracing is a solution that simplifies tracking user requests in distributed applications. It propagates a tracing context, relating to the original user request, along with the distributed nested requests. That allows for sampling, collecting and visualizing the trace information associated to each user request, at the cost of instrumenting the chosen communication libraries in order to propagate the tracing context. However, distributed tracing focuses primarily on collecting high-level user space events. Therefore, it will provide correct information about the flow and duration of requests, but cannot explain why some subrequests are too long when this is caused by operating system level contention – waiting on CPU, disk, network or mutexes.

Software tracing is a better-suited, powerful technique for collecting both kernel and user space events from a system. Software tracers record low-level events into trace files that can then be processed by dedicated analysis tools to diagnose bugs, contentions and latency. Because they collect precise, low-level events such as system calls, they are built for supporting a high throughput of events without disrupting the behavior of the target application or machine. While this is desirable for the type of analysis targeted, it cannot easily support the more costly tracing context propagation, like distributed tracers would do.

Getting at the same time the advantages of both distributed tracers and software tracers require to focus on hybrid approaches. In particular, a solution for collecting both distributed and kernel traces has to be devised, without sacrificing the performance of the target application. The traces have to be collected in a way that enables cross-level analysis, accurate and useful for performance characterization.

In this work, we used open source tools, LTTng (LTTng, 2018) as a software tracer, Jaeger (Jaeger, 2019) as a distributed tracer and Trace Compass (Eclipse, n.d.) as a trace analysis tool.

In this paper, we propose a new solution for fine-grained performance characterization of cloud applications using both distributed and kernel tracing. Our contributions are as follows:

— First, we propose and use a new and efficient architecture for combined trace collection and evaluate the impact on the performance of the target application.

— Second, we design and implement a new requests critical path analysis that can be used

for precise root cause finding.

In the second section, we review the related work. We describe in the third section the new proposed solution, from trace collection to analysis. Then, in the fourth section, we evaluate the overhead of our solution as well as the running time required for the analyses, and we illustrate the usefulness of our solution with a practical use case.

4.2 Related Work

Previous work on this subject includes studies on both performance analysis of distributed applications and low-level software tracing.

4.2.1 Performance Analysis of Distributed Applications

The most generic approaches to performance analysis require no change in the source code of the application. They can either try to infer what the optimal configuration for an application is, using machine learning (Chen et al., 2009), or assess its worst-case performance using controlled fault injection (Bennett and Tseitlin, 2012; Basiri et al., 2016). Fault injection can also be used to generate a database of labeled logs, and later compare the logs corresponding to an unknown failure to that database in order to find the closest matching fault (Pham et al., 2017). Leverich and Kozyrakis (2014) study machine-level scheduling algorithms for improving the use of server resources that can be suboptimal (Reiss et al., 2012).

Those techniques are limited when it comes to understanding bugs that are more complex or less frequent. The unmodified log files produced by an application can be used for request profiling (Aguilera et al., 2003) or extraction of independent user transactions (Zhao et al., 2014). In a similar way, log files can be processed using machine learning algorithms for latency root cause analysis (Nagaraj, Killian, and Neville, 2012; Kavulya et al., 2012; Tak et al., 2009). However, the techniques relying on log files are inherently limited by the quality and quantity of the data. Borderpatrol (Koskinen and Jannotti, 2008) gets around this limitation by dynamically loading a patched version of `libc` for more predictable quality of the log files, at the cost of a 10-15% performance penalty.

Application-specific approaches require a more or less direct instrumentation of the target application. Annotation techniques (Ma et al., 2017; Reynolds et al., 2006; Huang, Mozafari, and Wenisch, 2017) allow for better log data generation, while limiting the modifications to the source code. The annotations are usually destined to a compiler or profiler for generating instrumentation instructions. Dynamic instrumentation is also an alternative, when recompiling the target application has to be avoided. Examples of solutions that use this

technique include X-ray (Attariyan, Chow, and Flinn, 2012) for precise root cause analysis, or Fay (Erlingsson et al., 2012) for monitoring of distributed applications.

Distributed tracing focuses on end-to-end tracking of individual user transactions. Magpie (Barham et al., 2004) collects the events corresponding to remote procedure calls in order to identify causally related transactions and provide a consistent, end-to-end view of a user request. X-Trace (Fonseca et al., 2007) assigns a unique identifier to each request and propagates it along the data path, while keeping the event collection path distinct. Dapper (Sigelman et al., 2010) puts the emphasis on homogeneous systems and limiting the instrumentation to a small amount of communication and threading libraries. It allows for extensive data collection while preventing changes in the applications source code. Dapper also implements probabilistic sampling of requests to lower the performance impact of data collection. While trace collection naturally covers well the most frequent scenarios, Las-Casas et al. (2018) propose biased sampling to improve the diversity of the collected traces. Collection of distributed traces can be a basis for research on transaction profiling (Chanda, Cox, and Zwaenepoel, 2007), workflow-specific resource management (Mace, Bodik, et al., 2015) or live monitoring (Kaldor et al., 2017) and probing (Mace, Roelke, and Fonseca, 2018).

4.2.2 Low-Level Software Tracing

The techniques described above are particularly well adapted to distributed systems. While they focus on context propagation and tracing at scale, they can only collect high-level events, which limits in practice their ability to perform precise root cause analysis. At a lower level, LTTng is a kernel and user-space tracer for Linux with an emphasis on performance, with a measured impact of 2% for typical intense workloads (Desnoyers and M. R. Dagenais, 2006). According to Gebai and M. R. Dagenais (2018), it is the most performant kernel tracer for Linux. Unlike SystemTap (Prasad et al., 2005; Eigler and Hat, 2006), ftrace and eBPF (Fleming, 2017), it does not suffer from a significant increase in latency when used on multicore systems (Desnoyers and M. R. Dagenais, 2012). LTTng saves trace data to disk – or sends it through the network – for offline analysis and relies on the Common Trace Format (DiaMon, n.d.[b]), a standard format built for high trace event throughput. It offers a flight recorder mode that reduces the overhead by flushing the buffered events to disk only when the user requests a snapshot. Desfossez, Desnoyers, and M. R. Dagenais (2016) use this feature for efficient latency analysis of applications.

Low-level software tracing usually separates the event collection from the analysis that can be performed offline. LTTng traces can be analyzed to recover important metrics like CPU usage, with an arbitrary precision and without resorting to monitoring and fixed sampling

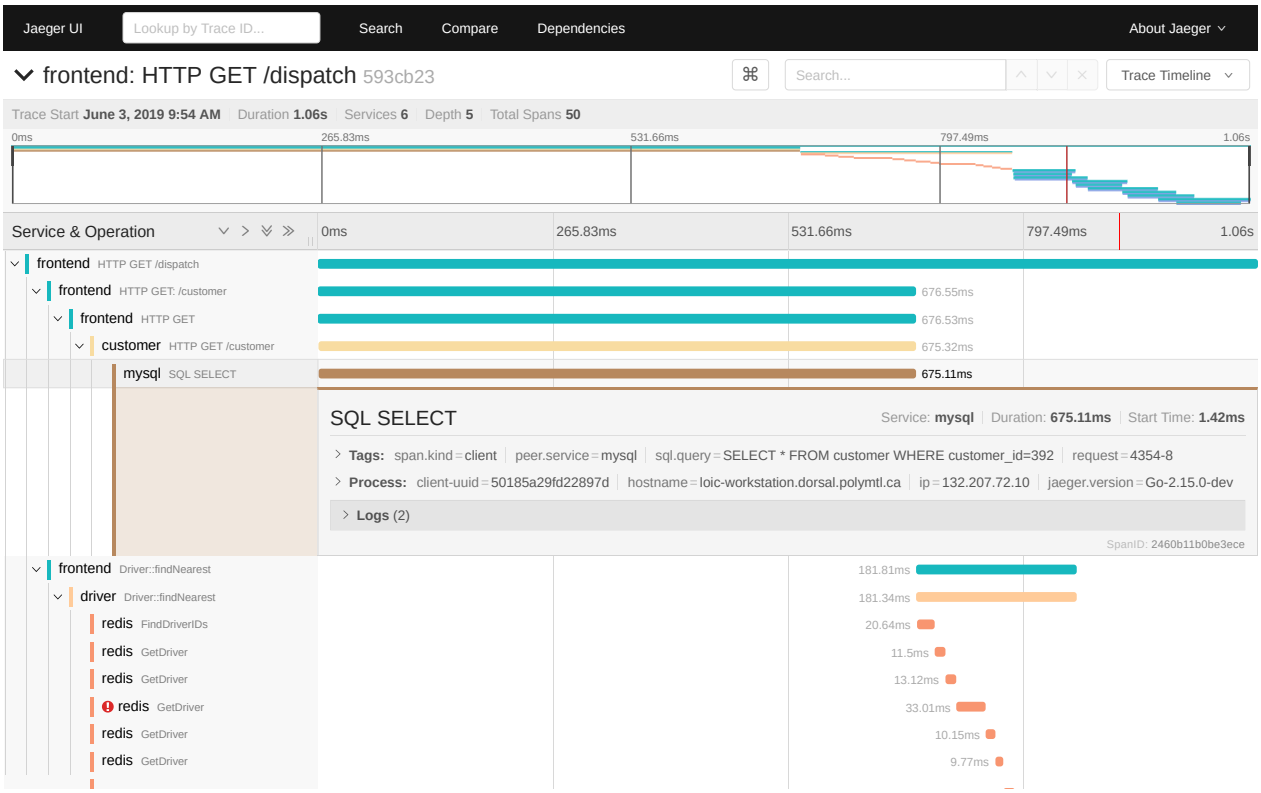

Figure 4.1 **Sample distributed trace shown by *Jaeger*.** The events corresponding to a user request can be collected and processed by the distributed tracer *Jaeger*. The view shows the complete flow of subrequests, along with the causal links between them, their duration and their log events.

rate (Giraldeau, Desfossez, et al., 2011). Trace Compass (Eclipse, n.d.) is a trace analysis tool that relies on an efficient, queriable state system (Montplaisir-Gonçalves et al., 2013) structure. It is built to compute and display the results of complex trace analyses, like a thread critical path analysis (Giraldeau and M. Dagenais, 2016).

Some research work tries to combine the flexibility of distributed tracing with the precision of kernel tracing. Ardelean, Diwan, and Erdman (2018) perform vertical context propagation to inject high-level request identifiers into the kernel. A system call is made every time a subrequest is started or finished in the target application, which can scale badly to high throughput applications. Sheth, Sun, and Sambasivan (2018) propose to patch LTTng so that it attaches a request context to every kernel event written in the trace file. The request context is injected into the kernel using a custom dynamic kernel module that adds the information to the current process structure `task_struct`. Both works emphasize the need for explicit synchronization, rather than simple interleaving, between kernel and distributed

tracing events.

4.2.3 Discussion

Previous work on distributed systems focuses on performance analysis using high-level information collected at the application level. In particular, distributed tracing proposes a standard way of tracking end-to-end requests to identify bottlenecks among subrequests or machines. As illustrated in Figure 4.1, distributed tracers provide the user with a unified view of the flow of a given request. However, they cannot explain why a particular subrequest is longer than expected, especially when this is caused by operating system level contention.

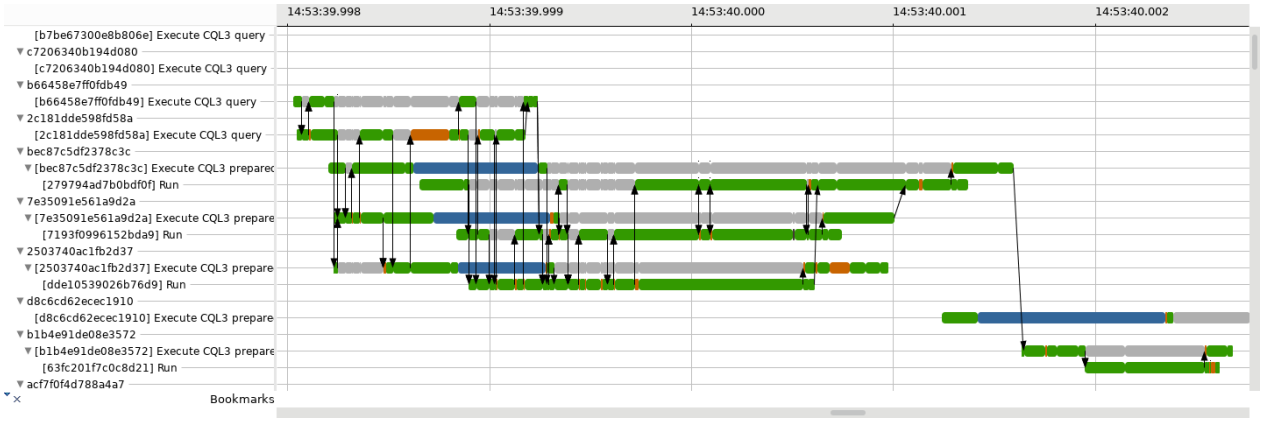

Figure 4.2 **View of our requests critical path analysis in Trace Compass.** Each state has a different color – green for `RUNNING`, orange for `PREEMPTED`, gray when blocked by another request, etc. –, and the arrows indicate how the different requests intersect in their respective critical path. Instead of showing a single request only, our view includes all the requests that happened in parallel.

The related work discussed above shows that kernel traces can be processed by specialized algorithms to identify complex bottlenecks and relate them to user processes and threads. Although some research work has focused on propagating distributed tracing events into kernel traces, no effort to the best of our knowledge has used distributed and kernel tracing altogether to improve root cause analysis.

Our objective is to propose an extended view of requests that is able to show and explain low-level contention. In particular, we propose a critical path analysis of requests, as shown in Figure 4.2, and explain our solution to obtain this analysis in the next section of this article.

4.3 Proposed Solution

The new proposed solution encompasses trace collection, analysis and visualization.

4.3.1 Instrumenting the Distributed Tracer

One of the key challenges in the problem studied is bridging the gap between:

— high-level, task-focused, microsecond-precise distributed traces on the one hand;

— and low-level, event-focused, nanosecond-precise kernel traces on the other hand.

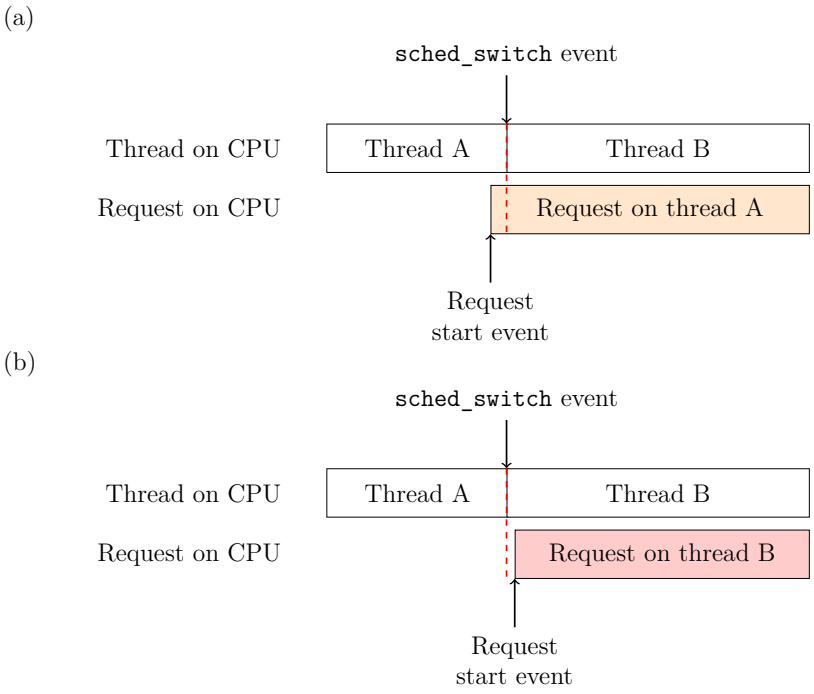

Figure 4.3 **Impact of the timestamps precision on trace analysis.** The role of an event-driven analysis is to reconstruct the state of a machine at any given time, using the sole trace events. Different event sequencing can yield different analysis results. In our case, insufficient precision on the distributed tracing events can impact the analysis, depending on whether the request starts (a) right before or (b) right after a scheduling event.

More specifically, integrating distributed traces to our analyses requires being able to improve the precision of the highest-level events. Let us suppose, for example, that we managed to get the event for the beginning of a distributed task right, at the nanosecond precision. Then we can use the events from the kernel trace corresponding to context switches between threads to infer which thread had actually started the request. In a different scenario with a lesser precision on the request start event, our analysis could be misled into inferring a different

thread information, and then computing the wrong critical path for that request. Figure 4.3 illustrates the two scenarios. The correlation between a request and the executing thread could be obtained by other means, like obtaining the thread identifier and including it as an event context field in the distributed tracer. This would require more changes to the distributed tracer and a higher overhead to obtain the thread identifier and store it in events.

The different techniques for traces synchronization discussed in Related work are based on vertical context propagation, at the cost of switching into kernel mode each time an event is emitted by the distributed tracer. Our solution avoids the overhead of vertical context propagation by collecting a minimal amount of synchronization events while staying in user-space mode. We use the LTTng tracer for collecting both kernel and user-space traces from applications. Because LTTng uses the same clock for kernel and user-space events, all the events emitted by a user application will be synchronized with the kernel trace, without any additional work needed. Therefore, the only application that requires instrumentation is the distributed tracer in-process agent. Our technique relies on the instrumentation of the distributed tracer itself with LTTng, so that it emits a user-space event each time a span is started or finished by the distributed tracer. The event carries the minimal information required for traces synchronization, namely the unique request identifier assigned by the distributed tracer. We implemented our solution on the distributed tracer Jaeger, more specifically for its Java and Go clients. Figure 4.4 describes how our solution can be deployed on a single host to collect at the same time (i) distributed traces using Jaeger, (ii) kernel traces using LTTng and (iii) user-space traces using LTTng. Listing 1 shows an example of synchronization events that are collected using our patched version of Jaeger.

It should be noted that some events will be present both in the distributed traces and the LTTng traces. This allows for more flexibility and adaptability to the targeted deployment. For example, in a large distributed application, it can be interesting to use Jaeger on every node but LTTng on specific chosen targets. We then get self-sufficient distributed traces, regardless of LTTng tracing, but benefit from the analyses of complementary traces, as described below, on specific nodes of interest.

4.3.2 Synchronizing the Traces

The Trace Compass trace analysis tool is able to read traces from both LTTng and Jaeger. The Jaeger traces describe every request as JSON data, including their start timestamp, duration, attached logs and parent request reference. Using every LTTng event that corresponds to the beginning or end of a request, we can correct the timestamps present in the

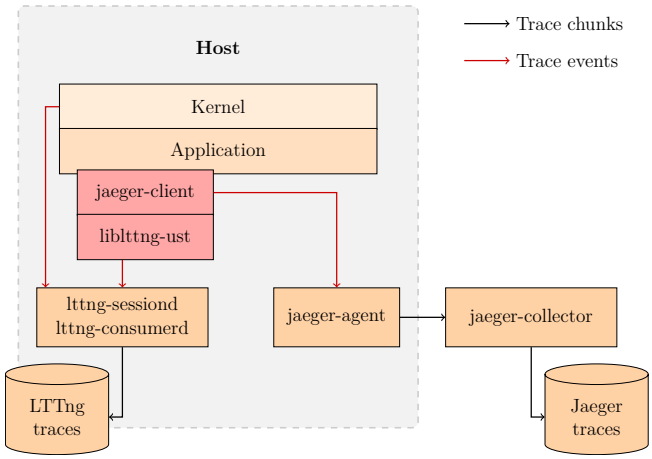

Figure 4.4 **Design of our trace collection solution.** Our solution targets a single host and is able to collect kernel traces using LTTng and distributed traces using Jaeger. The in-process Jaeger client is patched to emit LTTng user-space events and has to be compiled against `liblttng-ust`. The application code is unmodified, but it has to be compiled against the patched Jaeger client.

Jaeger traces into nanosecond-precise ones.

The challenging part is determining, for each request, the thread in which it is being executed at any given moment. It is fairly easy to infer the threads corresponding to the beginning and end of the request by using the scheduling events from the kernel trace. However, it is possible for a request to be handled by multiple threads in sequence. This is particularly true for asynchronous applications that rely on thread pools or pipelines to process queued tasks. For the sake of simplicity, we choose to associate a single thread to each sub-request, namely the thread that started the request. We then consider the critical path of that thread to be the critical path of the corresponding request. In the case of requests being migrated

Listing 1 *Sample synchronization events collected using our solution.* Our solution collects events in user-space mode for synchronization between distributed traces and kernel traces. Examples of such events are shown here, as displayed by the Babeltrace trace reader. Each event carries (i) nanosecond-precise absolute and relative timestamps, (ii) the name of the source server, (iii) the name of the event, (iv) the source CPU identifier and (v) the event payload.

```
[12:46:59.671901514] (+0.000001098) loic-workstation jaeger_ust:start_span: { cpu_id = 5 },
    { trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4421066052755260657 }
[12:46:59.672049028] (+0.000019061) loic-workstation jaeger_ust:start_span: { cpu_id = 5 },
    { trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4146236183556049419 }
[12:47:00.069919803] (+0.000000349) loic-workstation jaeger_ust:end_span: { cpu_id = 9 },
    { trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4146236183556049419 }
[12:47:00.492785641] (+0.000000988) loic-workstation jaeger_ust:end_span: { cpu_id = 1 },
    { trace_id_high = 0, trace_id_low = 4421066052755260657, span_id = 4421066052755260657 }
```

between threads, we would need more precise instrumentation to integrate the migrations to our analyses – see Conclusion and future work.

Conversely, we need to insure that a thread is only associated with a single request at any given time. When there are multiple possible requests associated with the same thread, we enforce that rule by marking the request started the most recently as being active. The full algorithm to associate threads to requests is shown in Algorithm 1.

Algorithm 1 *Pseudo-code for our synchronization algorithm.* The algorithm associates each thread to the sequence of requests it is actively handling. Its output is used by the extended critical path view discussed in subsection 4.3.4.

```
1:  Input
2:      requestEvents              list of requests
3:  Output
4:      allRequestsByTid           associates each TID to a mapping from intervals to a set of request UIDs
                                   corresponding to that interval
5:      activeRequestByTid         associates each TID to a mapping from intervals to a unique active request
                                   corresponding to that interval
6:  function MAIN
7:      lastTimestampByTid ← ()        ▷ associates each TID to the timestamp of the last processed
                                         event corresponding to that TID
8:      requestsStackByTid ← ()        ▷ associates each TID to a stack of request UIDs; the active
                                         request UID lies on top of the stack
9:      for e in requestEvents do
10:         HANDLEREQUESTEVENT(e, lastTimestampByTid, requestsStackByTid)
11:     end for
12: end function
13: function HANDLEREQUESTEVENT(e, lastTimestampByTid, requestsStackByTid)
14:     tid ← e.tid
15:     timestamp ← e.timestamp
16:     requestUID ← e.requestUID
17:     requestsStack ← requestsStackByTid[tid]
18:     lastTimestamp ← lastTimestampByTid[tid]
19:     if requestsStack is not empty then
20:         activeRequestByTid[tid][lastTimestamp → ts] ← top element of requestsStack
21:         allRequestsByTid[tid][lastTimestamp → ts] ← all elements of requestsStack
22:     end if
23:     lastTimestampByTid[tid] ← timestamp
24:     if e is a start event then
25:         push requestUID to requestStack
26:     else
27:         remove requestsStack[requestUID]
28:     end if
29: end function
```

4.3.3 Algorithm for Multi-Level Critical Path Analysis

Trace Compass includes a thread critical path analysis (Giraldeau and M. Dagenais, 2016). For any thread of interest to the user, the analysis computes its longest path of waiting after resources. It uses low-level events such as context switches or locking of mutexes to infer the cause of threads being blocked. The output of the analysis is a succession of the execution states that contribute to the latency of the target thread. Arrows in the output indicate blockage or wake-up links between threads.

Since we aim at extending the critical path analysis to user-level requests, it is key to understand how the critical paths of different threads interact so that we can understand how the critical paths of different requests interact as well. It is indeed possible for two different threads to share a state, i.e. contend for the same resource at the same time, in their respective critical paths. That happens for example when two threads are waiting on the same mutex held by a third thread; in that case, the critical paths of all three threads could reflect that contention by having a shared state, as explained in Figure 4.5(a).

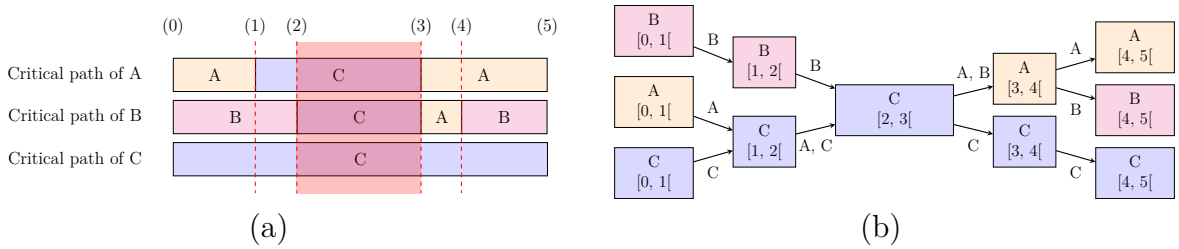

Figure 4.5 **Shared states between thread critical paths.** The critical paths of different threads can share one or more states (a). In this example, the thread C initially holds a mutex M and threads A, B and C are running. Each block indicates which thread is on the critical path of the thread of interest. The critical paths are built according to the following events sequence: (0) trace begins (1) A requests locking M, (2) B requests locking M, (3) C releases M and A locks M, (4) A releases M and B locks M, and (5) trace ends. The red overlay in subfigure (a) highlights the interval during which all threads have the same state in their respective critical path. Our critical path merging algorithm outputs the graph in subfigure (b). Each graph node indicates which thread the state refers to, as well as the time interval of the state. The critical path of a given thread can be recovered back from the graph by following the edges labeled with that thread name.

The critical path provided by Trace Compass is actually a linked list, in which each node is either:

— an interval that has a starting time, duration, a state and an associated thread identifier;

— an arrow that has a starting time, duration, a state, a source thread identifier and a target thread identifier.

Depending on the status of a thread and the resources it could be waiting after at a given time, Trace Compass will determine the state of a node to be `RUNNING`, `INTERRUPTED`, `PREEMPTED`, `TIMER`, `BLOCK_DEVICE`, `USER_INPUT` or `NETWORK`.

The previous work on thread critical path analysis is dedicated to kernel trace data only. The data structure chosen for the thread critical path is not adapted to cross-level analyses because it does not clearly identify states that are linked between different critical paths. The challenge in combining distributed traces with kernel traces in a critical path analysis lies in handling these shared states correctly.

We are addressing this problem by proposing a new algorithm that can identify contention on a shared resource between an arbitrary number of critical paths into a graph that includes them all. The goal is to have a single node in the output graph when it corresponds to contending for the same resource at the same time in different critical paths. A representation of the output of our algorithm, on the previous example in Figure 4.5(a), is given in Figure 4.5(b). More specifically, the nodes are Java objects for which we provide a simplified structure in Listing 2.

The two hash tables in the node structure allow for traversal of the graph in any direction. The key is the thread identifier of the critical path – an `Integer` object in Trace Compass – that connects the current node with the node associated with that key. The node also stores the node state – a `String` –, the thread identifier the state refers to, the start timestamp and the end timestamp.

A simplified version of our critical paths merging algorithm is shown in Algorithm 2. It gives a general idea of the approach that we are taking while staying relatively high-level. For the sake of completeness, it can be noted that most of the data structures that we actually use are of type `RangeMap` in Java. A `RangeMap` is basically a hash table in which keys are

Listing 2 *Simplified version of the Java data structure used for our critical path nodes.* A node from the graph obtained by merging critical paths has all the information needed to traverse the graph for any thread of interest.

```
class Node {
    private HashMap<Integer, Node> incomingEdges;
    private HashMap<Integer, Node> outgoingEdges;
    private String state;
    private Integer stateTid;
    private long startTs;
    private long endTs;
}
```

ranges, which allows for querying by intervals in logarithmic time. Those data structures get updated as the algorithm runs, so that their state consistently reflects the current data in the graph being built. From a high-level perspective, the algorithm loops over the states of the critical paths of interest to add each of them to the graph. Adding a state to the graph often requires splitting an existing state into two or three states – while updating the edges of the graph accordingly –, and then merging the duplicate states (i.e. contending for the same resource). The complete version of our algorithm keeps track of the head of the graph for each thread – that is, the first state in chronological order to be in that thread critical path –, allowing for targeted traversal of the graph.

4.3.4 Extending the Critical Path to the Requests

We propose an analysis that extends the critical paths of threads into critical paths of Jaeger requests. The goal is to have, for each request, a critical path that describes:

— how the critical path of the request intersects with critical paths of other requests: at a high level, that would mean that the request is waiting on another request because of a contention between their underlying threads;
— the states of the threads handling the request at a lower level.

We achieve this goal by putting together the algorithms shown in Section 4.3.2 and Section 4.3.3. After the traces are processed using these algorithms, we get the following elements:

— a graph representing the merged critical paths of all the threads of interest;
— for each thread, a data structure queryable by time range that maps time intervals to the unique identifier of the request active on that thread during that interval;
— for each thread, a data structure queryable by time range that maps time intervals to a set of unique identifiers of the requests processed by that thread during that interval.

Our analysis does a depth-first traversal of the critical paths graph. The critical paths of the requests are built incrementally by adding states to them while traversing the merged critical paths graph. For each state of the latter, we use our queryable map structures to determine the active request identifier corresponding to the time interval of the state. The state is simply copied to the critical path of that active request without modification. We then create a state in the critical path of each inactive request handled by the same thread; that state indicates that the request is currently blocked by the active request that we identified earlier.

Algorithm 2 *Simplified pseudo-code version of the critical paths merging algorithm.*

```
1: Input
2:     threadsOfInterest        list of integers
3: Output
4:     graphNodes               set of graph nodes
5: function MAIN
6:     for tidOfInterest in threadsOfInterest do
7:         criticalPath ← COMPUTECRITICALPATH(tidOfInterest)
8:         for state in criticalPath do
9:             MERGE(state)
10:        end for
11:    end for
12: end function
13: function MERGE(state)
14:    intersectingStates ← INTERSECT(graphNodes, state)
15:    for existingState in intersectingStates do
16:        MERGESTATES(existingState, state)
17:    end for
18: end function
19: function MERGESTATES(state1, state2)
20:    if state1 starts before state2 then
21:        newState ← CREATESTATE(state1.start, state2.start)
22:        REPLACELEFT(state1, newState)
23:        MERGESTATES(state1, state2)
24:        return
25:    else if state1 starts after state2 then
26:        newState ← CREATESTATE(state2.start, state1.start)
27:        REPLACELEFT(state2, newState)
28:        MERGESTATES(state1, newState)
29:        return
30:    else if state1 finishes before state2 then
31:        newState ← CREATESTATE(state1.start, state1.end)
32:        REPLACELEFT(state2, newState)
33:        MERGESTATES(state1, newState)
34:        return
35:    else if state1 finishes after state2 then
36:        newState ← CREATESTATE(state1.start, state2.end)
37:        REPLACELEFT(state1, newState)
38:        MERGESTATES(state2, newState)
39:        return
40:    end if
41: end function
42: function REPLACELEFT(state1, state2)
43:    state1.start ← state2.end
44:    for (tid, _) in state1.incomingEdges do
45:        state1.outgoingEdges[tid] ← state2
46:    end for
47:    state2.incomingEdges ← state1.incomingEdges
48:    state1.incomingEdges ← ()
49:    for (tid, _) in state2.incomingEdges do
50:        state1.incomingEdges[tid] ← state2
51:        state2.outgoingEdges[tid] ← state1
52:    end for
53:    state2.state ← state1.state
54: end function
```

4.3.5 Summarizing the Critical Paths

Because a kernel critical path is very detailed and low-level, it is sometimes useful to have a first look at a more synthetic piece of information. We propose a simple analysis that summarizes the critical path of a given request into the total time spent in various states: blocked by a concurrent request, waiting on CPU time, running, waiting on the network, etc. The summary fits into a pie chart and can help identify the root cause of a significant latency quickly.

Using the same approach, it is possible to extend our solution into additional views that rely on the analyses shown above. For example, one could find interesting to fetch the number of failed system calls in a distributed request, or the variation of the density of disk accesses during a transaction.

4.4 Evaluation

In this section, we evaluate the overhead of our trace collection solution, as well as the time required for running our analyses. We then demonstrate the usability of our implementation through a pratical use case.

We have implemented the analyses described above in Java, in the trace analysis tool Trace Compass. We have instrumented the Java and Go clients of Jaeger to add LTTng tracepoints for synchronization purposes. Our solution is then tested on two representative distributed applications.

1. HotROD [1] is a demo application for Jaeger that is already instrumented. It consists of multiple services including a web front end, a Redis server and a MySQL database. The user can make requests to order virtual car rides. We simulate loads of concurrent requests using Apache `ab` tool. We slightly modified HotROD to remove all simulated processing delays present in the demo application.
2. Cassandra [2] is a performance-oriented, NoSQL distributed database. We simulate loads consisting of concurrent write or read requests using the `cassandra-stress` tool. We have instrumented Cassandra so that it emits Jaeger events for each processed request. Because Cassandra is able to handle a very high throughput of requests, it is likely to generate a lot more tracing data and thus be more impacted by our trace collection solution.

1. `https://github.com/jaegertracing/jaeger/tree/master/examples/hotrod`
2. `http://cassandra.apache.org/`

In both cases, the application, the Jaeger services and the LTTng daemon execute on the same physical machine to simplify the evaluation. The machine is equipped with an Intel® Core™ i7-7820X processor (8 physical cores, 16 virtual cores, 3.60Ghz base frequency) and 32GB of RAM.

4.4.1 Trace Collection Overhead

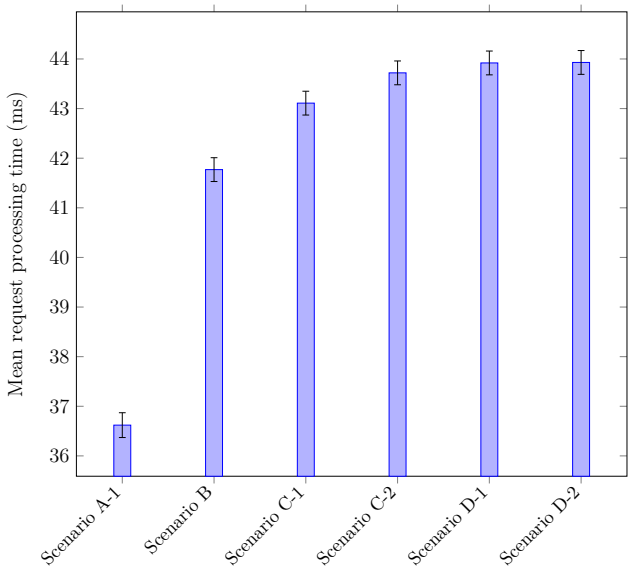

Figure 4.6 **Request processing times for HotROD using different tracing scenarios.** The two configurations corresponding to our solution are scenarios C-1 and C-2, while the baseline – tracing with Jaeger only – is scenario B. For reference, scenario A-1 corresponds to disabling all tracing. Scenarios D-1 and D-2 are two configurations for the fake syscall technique that we implemented for comparison.

We show that our solution adds a reasonable overhead, when compared to only collecting Jaeger traces in an application, while generating a significant amount of kernel and synchronization events. We measure the average request processing time for both HotROD and Cassandra in the five following scenarios.

A-1. No tracing Both Jaeger and LTTng are turned off: no traces are collected.

B. Jaeger only This is the baseline to which our solution will be compared: only Jaeger traces are collected. Sampling is off for HotROD – meaning that all the requests are traced –, while 10% only of the requests are sampled for Cassandra. This is to avoid a huge overhead that would be very unlikely in a production setting for Cassandra.

C-1. Full (snapshot mode) This is the first configuration for our solution. Jaeger traces are collected, with a sampling configuration identical to the previous scenario. LTTng

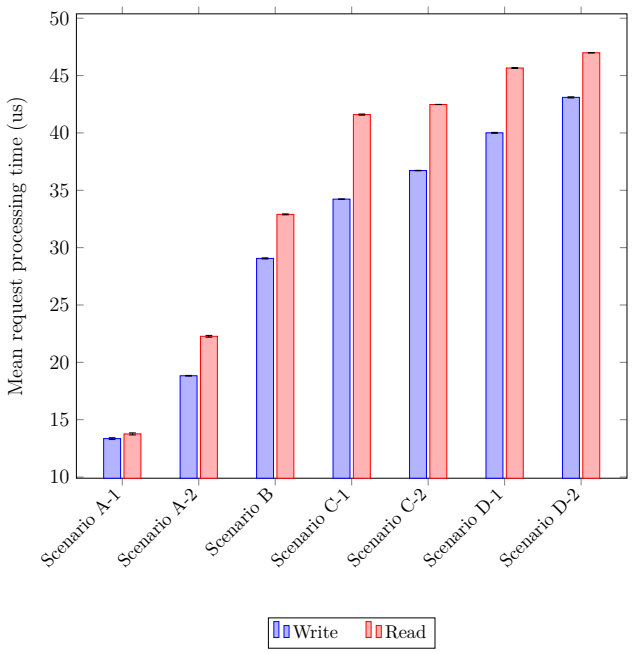

Figure 4.7 **Request processing times for Cassandra using different tracing scenarios.** The two configurations corresponding to our solution are scenarios C-1 and C-2, while the baseline – tracing with Jaeger only – is scenario B. For reference, scenario A-1 corresponds to disabling all tracing and scenario A-2 to using the built-in request logging feature in Cassandra. Scenarios D-1 and D-2 are two configurations for the fake syscall technique that we implemented for comparison. For each scenario, we distinguish between read and write workloads.

is also running to collect synchronization events, and a subset of kernel events required for most analyses in Trace Compass. LTTng is running in *snapshot* mode, meaning that the events are kept in circular buffers in memory until the user requests them to be flushed to a trace file. This is a production-friendly mode of LTTng that allows for collecting an actual trace only when an anomaly is detected. The performance is better than in standard mode, and the size of the trace files can be controlled more easily. In this scenario, we make sure to never trigger a trace flush, so as to simulate passive events recording with no anomaly. Because we never write the events to disk, they keep being overwritten by newer ones when the circular buffer is full.

C-2. Full (standard mode) This is the second configuration for our solution. It is identical to the first one, except that LTTng is run in standard mode. The events that are collected are all written to a trace file.

D-1. Fake syscall (snapshot mode) We implemented the solution proposed by Ardelean, Diwan, and Erdman (2018). Instead of instrumenting the distributed tracer with dedicated synchronization events collected in user space, we modify it so that it emits a

`getpid` system call for every request, beginning or end. Unlike a normal `getpid` system call, we add the request unique identifier as an argument that will be written to the trace buffers. LTTng is running in snapshot mode.

D-2. Fake syscall (standard mode) This scenario is identical to the previous one, but LTTng is running in standard mode.

We include an additional scenario for Cassandra only:

A-2. Integrated tracing Cassandra is configured to trace all the requests, using its builtin logging feature. All the tracing events are printed to a file.

The average request processing time is an important metric because it directly impacts the throughput of the target application. For distributed applications that are used in production, it is key that the trace collection does not reduce significantly the throughput of requests.

Results for HotROD

Table 4.1 **Overhead of our trace collection solution for HotROD.** The overhead of our solution compared to the baseline is below 4% when using the snapshot mode in LTTng, and around 5% when using the standard mode.

Reference \ Scenario	B	C-1	C-2	D-1	D-2
A-1	14.1±0.6%	17.7±0.6%	19.4±0.7%	19.9±0.7%	20.0±0.7%
B (baseline)		3.2±0.6%	4.7±0.6%	5.2±0.6%	5.2±0.6%
C-1				1.9±0.6%	
C-2					0.5±0.5%

We executed a total of 10,000 requests using 10 client threads for each of the above scenarios. The average processing times are summarized in Figure 4.6. The relevant overheads between scenarios are given in Table 4.1.

Even when tracing all the requests with Jaeger and LTTng, the overhead of our solution compared to tracing with Jaeger only is about 5%, and is below 4% when using the snapshot mode from LTTng. This overhead allows using our solution in a production environment without a huge impact on the application throughput. The fake syscall technique performs slightly worse than our solution, though without adding a significant penalty.

Results for Cassandra

The tool `cassandra-stress` is able to send read or write requests to a Cassandra server. Because they lead to workloads of different nature, it is interesting to benchmark our solution

Table 4.2 **Overhead of our trace collection solution for high-throughput read workloads in Cassandra.** The overhead of our solution compared to the baseline is below 18% when using the snapshot mode in LTTng, and below 27% when using the standard mode.

Reference \ Scenario	A-2	B	C-1	C-2	D-1	D-2
A-1	41.0±1.1%	117.7±1.7%	156.4±1.7%	175.0±1.9%	199.7±2.2%	222.8±2.4%
A-2		54.3±0.5%	81.8±0.4%	95.0±0.4%	112.5±0.6%	128.9±0.7%
B (baseline)			17.8±0.2%	26.3±0.3%	37.7±0.4%	48.3±0.4%
C-1					16.9±0.17%	
C-2						17.4±0.20%

Table 4.3 **Overhead of our trace collection solution for high-throughput write workloads in Cassandra.** The overhead of our solution compared to the baseline is below 27% when using the snapshot mode in LTTng, and around 29% when using the standard mode.

Reference \ Scenario	A-2	B	C-1	C-2	D-1	D-2
A-1	61.8±1.7%	139.0±1.9%	202.3±2.4%	208.7±2.2%	231.8±2.6%	241.4±2.5%
A-2		47.7±0.7%	86.8±0.9%	90.7±0.8%	105.0±1.0%	111.0±0.9%
B (baseline)			26.5±0.3%	29.1±0.3%	38.8±0.3%	42.8±0.3%
C-1					9.8±0.23%	
C-2						10.6±0.07%

for both. We executed a total of 1,000,000 read requests, then 1,000,000 write requests, using 20 client threads for each scenario. The average processing times are summarized in Figure 4.7. The relevant overhead for each scenario is given in Table 4.2 for read loads and in Table 4.3 for write loads.

This time, the performance impact of our solution is more significant, especially because Cassandra has a very high throughput and we are sampling 10% of the requests. Compared to tracing with Jaeger only, our solution adds a lot of kernel and synchronization events to the trace, at the cost of a 18-27% performance penalty. Using our solution in a production setting would require lowering the sampling by several orders of magnitude, and using LTTng in snapshot mode instead of standard mode.

Because of the throughput of tracing events that trigger a lot of switching into kernel mode, the performance is even worse with the fake syscall technique, that adds a 17-18% overhead to our solution.

4.4.2 Trace Analysis Duration

It is interesting to get a sense of the time spent in analyzing the traces. We created a set of test traces of different sizes and contents using `cassandra-stress`. We then run our analyses

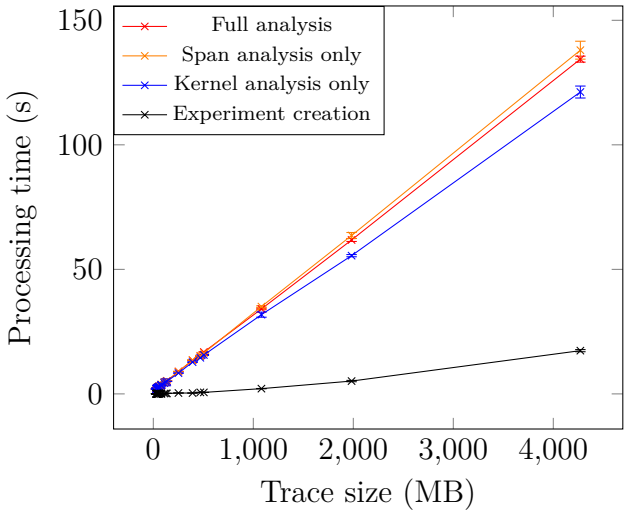

Figure 4.8 **Execution time of our critical path analysis of requests.** Considering the test traces used, our analysis seems to scale linearly with the trace size. The execution time is very reasonable for moderately large traces.

several times on each trace and record the average running time and confidence interval around the average, as given by the Java JUnit testing tool. The results are described in Figure 4.8.

Interestingly, the time required to perform our extended critical path analysis alone is very close to the full analysis time – that includes the kernel analysis from Trace Compass, along with our analysis –, meaning that the parallelization in Trace Compass is almost perfect. The full analysis takes about 15 secondes for traces as big as 500MB – which corresponds to 20-30 seconds of tracing data.

4.4.3 Use Case: Logical CPU Pressure Using Control Groups

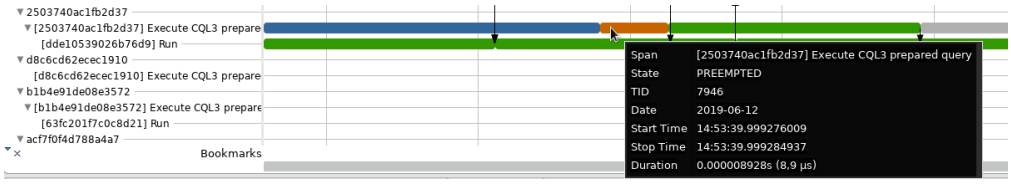

Figure 4.9 **Use case: tooltips.** The tooltips in our analysis allow for getting more information about a given critical path state.

We demonstrate the usefulness of our solution on a practical use case, for which we generate a precise fault in a controlled environment. We show how our implementation can help identify

the faults by combining distributed tracing events and kernel events in our analyses.

Our use case involves a bad configuration of a `cpu` control group. Control groups are a very popular mechanism, being an important building block for Linux containers. They allow for limiting the logical or physical resources allocated to a given group of processes. The `cpu` control group, for example, is able to limit the CPU usage for a set of processes to a fixed number of cycles per second. This use case is especially interesting since CPU contention is a frequent source of problems and is difficult to diagnose without kernel tracing. Furthermore, using control groups to create contention is convenient for experimentation while being more difficult to diagnose than more common sources of contention (e.g. competing threads).

We run Cassandra and put all its threads into a `cpu` control group, while collecting trace events with Jaeger and LTTng. As we put Cassandra under a constant load, we limit the CPU usage for that control group to 1% only of the available CPU time, and then waive the limit a few seconds later.

When we open the trace in Trace Compass and run our analyses, we end up getting the critical path for all the requests. Tooltips provide more information about a given state – see Figure 4.9.

We can easily identify the abnormal latency in the trace. Figure 4.10 shows that some requests that are identical have very different processing times. During the time where the latency is huge – which we know corresponds to the control group limit that we set –, we can observe in Figure 4.11 that the threads in Cassandra seem to share very scarce CPU resources. Figure 4.12 shows that the CPU becomes significantly underused when the control group limit kicks in. When we zoom in on a longer request – see Figure 4.13 –, we can see that it gets preempted every 100ms for the same amount of time. This information is also available from our summarized critical path view shown in Figure 4.14.

Further investigation can help diagnose the problem. The Resources view in Trace Compass clearly shows that the threads are preempted but not replaced on the CPU, as shown in Figure 4.15. In other words, the threads in Cassandra are not preempted because a higher priority process has to be scheduled in. The scheduler event corresponding to Figure 4.15, as displayed by the Events view in Trace Compass, is shown in Table 4.4.

Table 4.4 **Use case: scheduling event for a worker thread.** The worker thread is replaced by the virtual process `swapper`, which indicates that the CPU is becoming idle. The state of the worker thread is left to 0 – meaning it could potentially be running.

CPU	Event type	Contents	TID	Prio	PID
2	sched_switch	prev_comm=java, prev_tid=7949, prev_prio=20, prev_state=0, next_comm=swapper/2, next_tid=0, next_prio=20	7949	20	5322

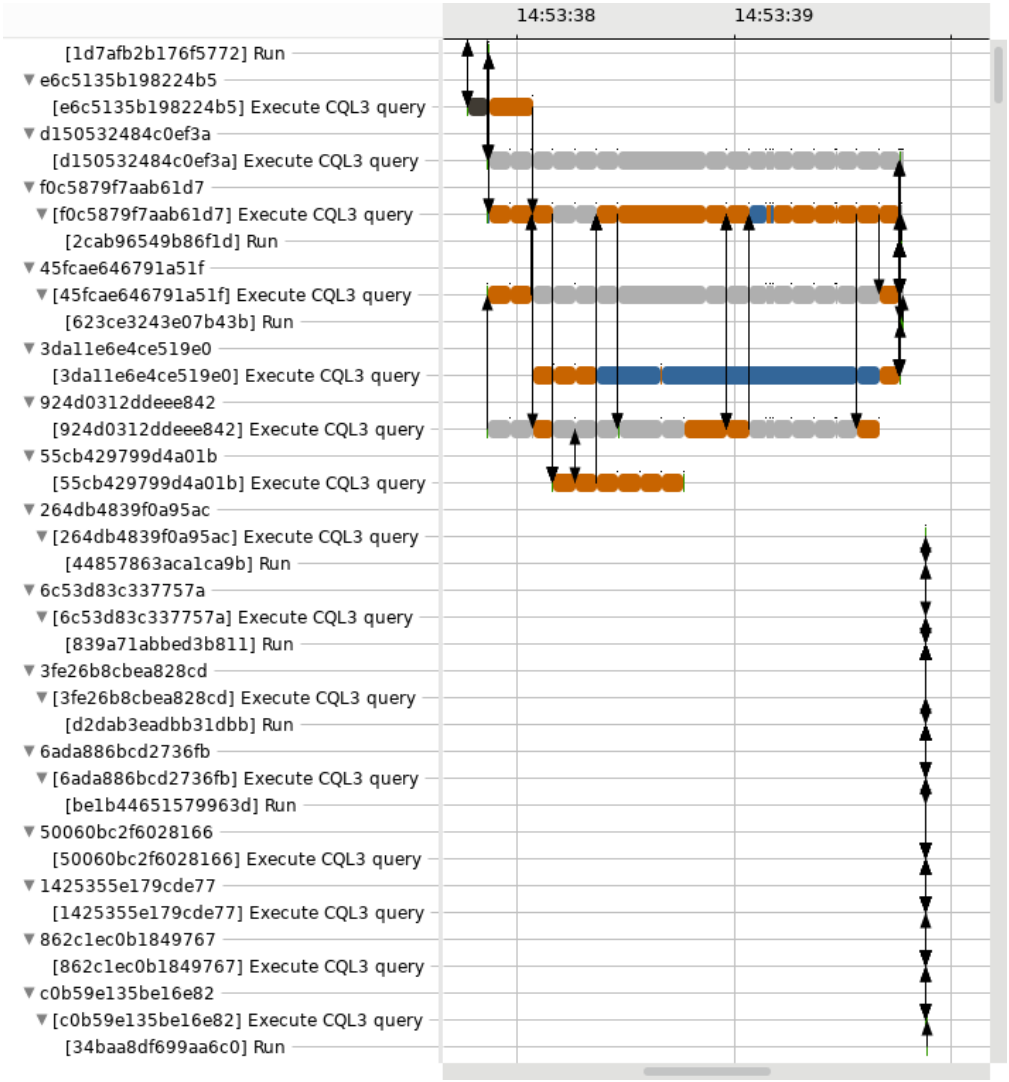

Figure 4.10 **Use case: identifying the latency.** The requests are all equivalent, but the ones on the left take 2 seconds to process, as opposed to about 5ms for the ones on the right. The view shows that the slow requests spend most of their time waiting on CPU or on another request.

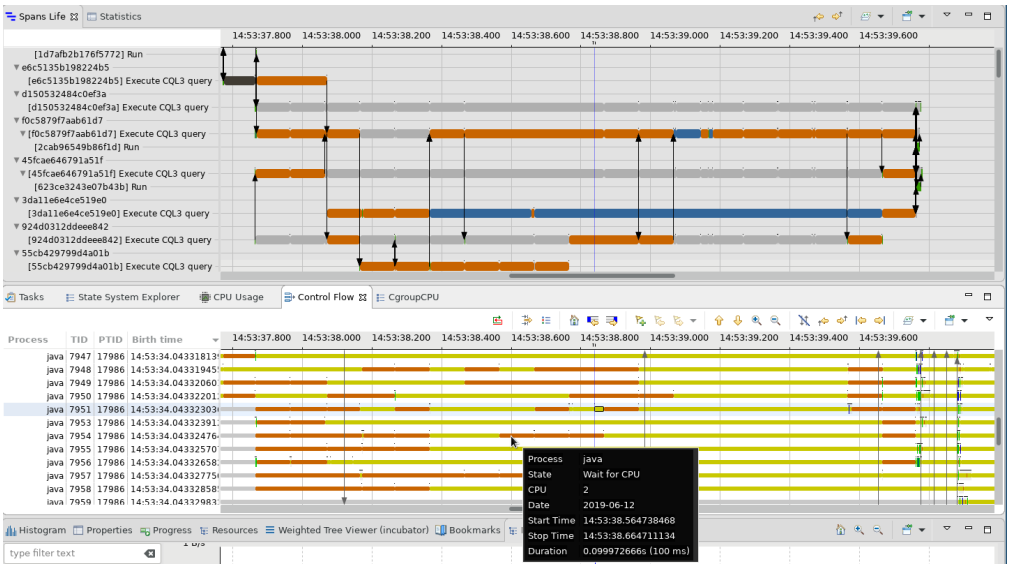

Figure 4.11 **Use case: CPU time sharing between worker threads.** The Resources view in Trace Compass shows that the threads in Cassandra all share scarce CPU resources. They are waiting on CPU or blocked most of the time.

In a similar fashion, the kernel events corresponding to the control group limit being applied or waived can be identified. Along with the analyses and views included in Trace Compass, our solution allows for in-depth root cause finding, since it relies on detailed kernel events collection.

4.4.4 Discussion

Our solution has a variable overhead, depending on the nature of the target application. The overhead measured is about 5-6% for HotROD – with 100% of the requests being sampled –, and close to 30% for Cassandra – for a sampling frequency of 10%. The shorter duration of requests for Cassandra is impacted more by the added kernel tracing than the longer duration for HotROD requests. It is worth mentioning than a sampling frequency of 10% for high-throughput applications is acceptable for testing and benchmarking, but unreasonable for production environments. A good trade-off between performance and trace collection can be achieved through lower sampling frequencies.

Our full critical path analysis for requests requires about 15 seconds, for processing moderately large traces of 20 to 30 seconds. This is highly acceptable when analysing a small number of problematic requests in order to diagnose a problem. The use case that we presented demonstrates the interest of multi-level trace data analysis. We argue that our analysis explains the root cause of actual latencies in a way that distributed tracers alone

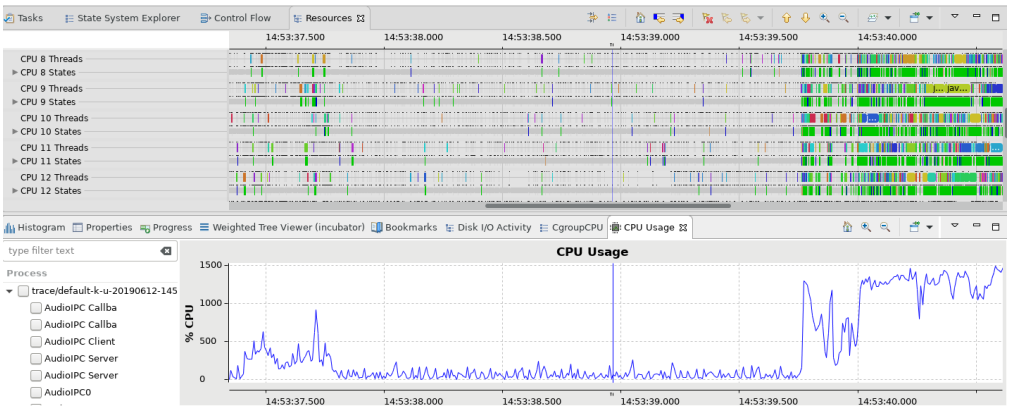

Figure 4.12 **Use case: suboptimal usage of the CPU.** Heavy CPU limit can lead to the machine being underused. Once the CPU limit is waived, the usage is back to normal.

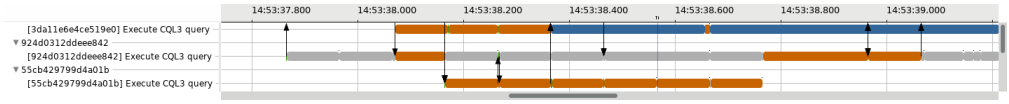

Figure 4.13 **Use case: pattern of preemption.** Request `55cb429799d4a01b` takes about 600ms to process, but shows a pattern of recurring preemption, every 100ms.

simply cannot achieve.

4.5 Conclusion and Future Work

In this paper, we explored how to combine distributed tracing with kernel tracing to better characterize the performance of distributed applications. We proposed a solution for collecting and synchronizing the trace events, using Jaeger as a distributed tracer and LTTng as a kernel and user space tracer. Our solution for propagating the context from the distributed tracer into the kernel traces relies solely on the instrumentation of Jaeger in user space. We show that extensive trace collection can be achieved without patching the application. The overhead is reasonable for average-throughput applications, and can be kept low enough for production environments on high-throughput applications, provided that trade-offs are made on the requests sampling frequency. Beyond this, we proposed analyses that compute the critical path of distributed requests and implemented them in Trace Compass. We showed that the analyses could be run in a couple of seconds for moderately large traces. We illustrated how our analyses can provide useful information about precise kernel events or contentions, and relate that information to the latency of given requests.

Further work should focus on integrating our solution to existing monitoring dashboards

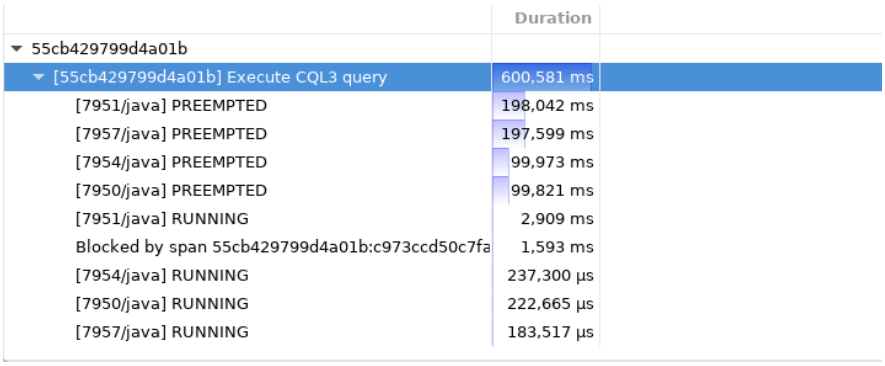

Figure 4.14 **Use case: summary of a critical path.** The critical path summary shows that request `55cb429799d4a01b` spends most of its time in the `PREEMPTED` state.

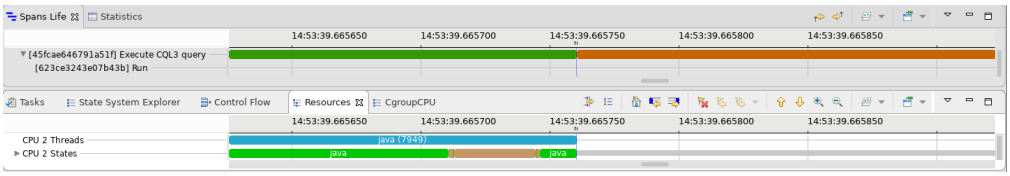

Figure 4.15 **Use case: preemption without replacement.** The Resources view in Trace Compass shows that the `java` thread used by Cassandra is preempted, but not replaced.

for better usability. Trace collection under the form of snapshots could be triggered when anomalies are detected by the monitoring application, and lead to automated processing and reporting back to the dashboard for manual analysis. In addition, user-level task schedulers, work queues and thread pools are getting more popular, which makes the analysis based on kernel scheduling events more difficult. Our algorithms will be accurate only if we are able to track requests being moved between threads, and conversely determine which request is being actively handled by a given thread. That will imply better support by the distributed tracers for multithreaded frameworks and languages.

4.6 Acknowledgements

The authors of this paper are grateful to Ciena, EfficiOS, Ericsson, Google, the Natural Sciences and Engineering Research Council of Canada (NSERCC) and Prompt for their financial support.

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous revenons sur les contributions et résultats présentés dans l'article précédent. Nous proposons également des pistes de réflexion dans la continuité de notre travail.

5.1 Surcoût de notre solution

Notre solution de collecte de traces présentée dans l'article semble montrer un surcoût très variable selon l'application ciblée. Dans le cas de *HotROD*, le surcoût est de l'ordre de 5-6% alors qu'il approche les 30% pour l'application *Cassandra*. Deux constats permettent d'expliquer cette différence ainsi que l'importance du surcoût.

Différence de débit des requêtes entre les deux applications Contrairement à l'application *HotROD*, développée uniquement à des fins de démonstration du traceur *Jaeger*, *Cassandra* est une application centrée sur la performance et très largement utilisée dans des systèmes répartis en production. Cette dernière traite donc un débit de requêtes beaucoup plus important. À titre de comparaison, d'après nos mesures effectuées en l'absence totale de traçage, une requête *HotROD* est traitée en moyenne en $37ms$ alors qu'une requête *Cassandra* ne prend que $14\mu s$, soit environ 2600 fois moins de temps. Le même surcoût associé au traçage d'une requête va donc représenter un pourcentage plus important du temps de cette requête dans une application aussi performante que *Cassandra*.

Différence de langage entre les deux applications et le traceur *LTTng* Notre technique consiste à instrumenter la librairie client du traceur *Jaeger* avec *LTTng*. *HotROD* étant écrit en Go et *Cassandra* en Java, il a fallu instrumenter les deux librairies de *Jaeger* différemment. Une difficulté supplémentaire est que le traceur *LTTng* n'est pas nativement compatible avec les langages Go et Java. Nous avons donc utilisé l'agent *LTTng* écrit en C pour le client Go de *Jaeger*, en utilisant la librairie cgo qui permet d'exécuter du code C depuis un programme Go. En raison notamment des conventions d'appel de fonction qui sont différentes entre C et Go, le surcoût d'un appel de fonction C depuis du code écrit en Go est non négligeable, de l'ordre de 200ns (Carruthers-Smith, 2018). Pour Java, *LTTng* propose une technique qui permet d'émettre des événements sous forme de chaînes de caractères. La performance n'est donc pas aussi optimisée qu'elle pourrait l'être en utilisant directement le traceur *LTTng* développé

pour C.

Il est néanmoins important de limiter au mieux le surcoût associé au traçage. En plus de l'impact sur le débit des requêtes traitées, qui n'est pas souhaitable en production, un surcoût trop important peut potentiellement changer le comportement du système au point de faire disparaître les problèmes que l'on cherche à tracer pour mieux les comprendre. Or, il convient de souligner que les mesures effectuées dans l'article correspondent à un échantillonnage de 10% des requêtes – c'est-à-dire que 10% des requêtes sont tracées en moyenne – pour *Cassandra*, et 100% des requêtes pour *HotROD*. Un moyen simple de limiter le surcoût associé au traçage est de diminuer la fréquence d'échantillonnage de plusieurs ordres de grandeur. En pratique, même pour des fréquences aussi basses que 0.1% – qui est la valeur par défaut choisie par *Jaeger* –, un grand volume de requêtes nous garantira de récolter assez de traces pertinentes à analyser.

Un autre moyen de limiter le surcoût associé au traçage *LTTng* est le mode de prise d'instantané fourni par *LTTng*. Il permet de conserver les événements collectés dans des tampons circulaires en mémoire, pendant quelques secondes, et de ne les écrire dans des fichiers de trace que sur demande de l'utilisateur. Cela permet de diminuer l'utilisation du processeur et du disque par *LTTng*, et de ne créer des fichiers de trace que lorsque c'est nécessaire, par exemple lorsqu'une latence est détectée. Nous avons utilisé cette technique avec succès pour collecter les traces noyau dans le cas d'utilisation décrit dans l'article. Pour ce faire, nous avons modifié le traceur *Jaeger* afin qu'il envoie une requête de prise d'instantané lorsqu'il détecte une requête dont la durée dépasse un seuil fixé. La requête est envoyée à un *socket* UNIX sur lequel écoute un utilitaire que nous avons écrit, et qui déclenche une prise d'instantané *LTTng*. Cette technique peut être étendue aisément au cas où la prise d'instantanée est décidée par une machine distante – voir la section 5.3 plus bas.

5.2 Échantillonnage des requêtes

Comme nous venons de le voir, un échantillonnage plus faible des requêtes permet de réduire l'impact du traçage sur le système. En revanche, il pose un problème avec l'analyse de chemin critique que nous avons proposée. Pour être pertinente, cette analyse doit considérer conjointement toutes les requêtes qui ont lieu en même temps qu'une requête cible. L'échantillonnage probabiliste proposé par *Jaeger* ne permet pas de garantir que toutes ces requêtes seront tracées, car la décision d'échantillonnage est prise individuellement pour chacune. De la même manière, des techniques d'échantillonnage comme celle proposée par Las-Casas et al. (2018) ne considèrent pas des groupes de requêtes, mais bien des requêtes prises sé-

parément. Une façon pour les traceurs distribués de contourner le problème pourrait être d'échantillonner sur des intervalles de temps – tout en conservant les mêmes heuristiques – plutôt que sur des requêtes prises individuellement. Cela ne modifierait pas notre solution de collecte de trace, mais garantirait le bon fonctionnement de nos analyses.

5.3 Intégration des analyses dans des outils standard de monitorage

Les applications déployées dans le nuage sont régulièrement contrôlées à l'aide d'outils de monitorage. Un panneau de contrôle permet à l'utilisateur de s'assurer en temps réel que la latence ou encore la charge sur les serveurs sont satisfaisantes. Ces outils ont en revanche peu d'interactions avec des outils spécialisés de collecte et d'analyse de traces.

Il serait intéressant d'intégrer progressivement des analyses telles que celle que nous avons proposée dans de tels outils. En particulier, un outil de monitorage peut permettre de détecter un problème grâce à des changements significatifs de certaines métriques. Quand un tel problème est détecté, il peut donner lieu à une prise d'instantané sur le serveur concerné, puis à une analyse dont le résultat serait affiché dans l'interface de l'outil de monitorage. L'outil Trace Compass est d'ores et déjà utilisable en tant que serveur d'analyse, et il propose un protocole – le *Trace Server Protocol* – qui permet l'envoi de traces et la réception des résultats d'analyse. Il reste encore à développer les vues correspondantes dans les interfaces des outils de monitorage ciblés, ainsi que l'infrastructure distribuée de collecte automatique des traces et de détection des problèmes.

5.4 Amélioration de l'analyse de chemin critique des requêtes

Nous avons dû faire quelques hypothèses dans le cadre du travail présenté dans l'article. En particulier, nous avons considéré qu'une requête n'était jamais traitée par plus d'un seul fil d'exécution. Or, il est courant pour une application de traiter des tâches en file d'attente à l'aide de fils d'exécution, qui se servent à tour de rôle dans cette file d'attente lorsqu'ils sont libres. Une tâche associée à une requête peut donc être créée par le fil d'exécution chargé de mettre les requêtes en file d'attente, puis traitée par un fil d'exécution différent. Une instrumentation des librairies de gestion de tâches serait nécessaire pour que notre analyse puisse prendre en compte ce genre de cas. De même, certains langages comme Go proposent nativement une fonctionnalité de fils d'exécution dans l'espace utilisateur. Les événements d'ordonnancement correspondant à ces fils d'exécution ne sont pas visibles dans des traces noyau, et ce cas particulier pourrait faire l'objet d'un effort d'instrumentation pour s'assurer de la fiabilité des analyses.

Une seconde hypothèse que nous avons faite est que la requête activement traitée par un fil d'exécution est la requête créée la plus récemment, parmi celles qui sont affectées à ce fil d'exécution. Ce n'est pas forcément vrai puisque l'ordre de création des tâches peut ne pas correspondre à leur ordre de traitement. Plutôt que se reposer sur une heuristique simple, comme nous l'avons fait, pour déterminer la requête à considérer sur un fil d'exécution, le mieux est d'annoter ou d'instrumenter le code de l'application de manière à ajouter à la trace l'information manquante.

Si ces éléments ne remettent pas en cause la validité théorique de nos analyses, ils montrent que le travail des développeurs d'applications, de librairies et de langages est important pour améliorer la transparence de leurs outils. Sans une instrumentation suffisante des différentes couches logicielles sur lesquelles repose une application, il est difficile de proposer des analyses pertinentes et utilisables.

CHAPITRE 6 CONCLUSION

Nous concluons ce mémoire par une synthèse des travaux réalisés dans le cadre de ce projet de recherche. Nous analysons ensuite les limitations de la solution avant de proposer des pistes de réflexion pour l'améliorer.

6.1 Synthèse des travaux

Dans le cadre de ce projet, nous avons proposé et implémenté une solution permettant de collecter et d'analyser les traces correspondant à l'exécution d'une application distribuée, afin de mieux caractériser sa performance. Les quatre paragraphes suivants répondent, dans l'ordre, aux questions de recherche soulevées dans l'introduction de ce mémoire.

Dans un premier temps, nous avons conçu un mécanisme permettant de collecter conjointement les traces distribuées et les traces noyau sur une machine cible. Notre implémentation repose sur l'utilisation de *LTTng* comme traceur noyau, et sur l'instrumentation avec *LTTng* du traceur distribué *Jaeger*. La technique permet d'obtenir des traces de nature différentes, synchronisées, et avec un surcoût raisonnable pour des applications qui traitent un débit de requêtes modéré. Dans le cas d'applications plus intensives, le surcoût peut être maintenu assez faible au besoin en ne collectant qu'une fraction des nombreux événements associés aux requêtes.

Dans un deuxième temps, nous avons proposé puis implémenté dans l'outil *Trace Compass* un algorithme permettant de lire et de synchroniser les traces collectées. Cette étape est essentielle pour associer chaque requête – une information de haut niveau – à un fil d'exécution noyau – une information de bas niveau – grâce notamment aux événements issus de l'instrumentation de *Jaeger* par *LTTng*.

Ensuite, nous avons conçu une analyse qui calcule le chemin critique de chacune des requêtes rencontrées dans la trace étudiée. Ce chemin critique est calculé à partir du chemin critique de tous les fils d'exécution concernés par les requêtes, et permet de savoir à tout moment quel fil d'exécution, quelle ressource ou quelle requête concurrente est en train de bloquer la requête analysée.

Enfin, nous avons implémenté l'analyse de chemin critique des requêtes dans l'outil *Trace Compass*, et proposé une vue pour afficher le résultat de cette analyse. L'utilisateur peut donc voir le chemin critique de toutes les requêtes sous la forme d'une succession d'états, avec des flèches permettant de suivre le chemin critique d'une requête en particulier lorsque

ce chemin critique est bloqué par d'autres requêtes. La vue permet à l'utilisateur d'identifier rapidement et d'expliquer les zones de latence présentes dans la trace.

Au-delà des objectifs de recherche énoncés dans l'introduction de ce mémoire, notre travail a permis de vérifier la validité d'approches hybrides de collecte et d'analyse des traces. L'analyse du chemin critique des requêtes est un exemple complet qui combine des informations collectées à différents niveaux de la pile logicielle. Surtout, l'approche proposée est flexible puisqu'elle ne se substitue pas complètement au traçage distribué. Il est possible d'envisager un déploiement qui repose de manière autonome sur des outils comme *Jaeger*, et d'y déployer *LTTng* en complément, comme nous l'avons proposé, sur quelques machines choisies.

6.2 Limitations de la solution proposée

Une première limitation qui nous est apparue pendant ce projet de recherche est la difficulté de trouver des applications distribuées prêtes à être instrumentées. Il fallait notamment trouver une application suffisamment complexe pour que l'analyse proposée présente un intérêt. Parmi les potentiels candidats, comme *Apache Spark* ou encore *Elasticsearch*, la plupart n'utilisent pas de librairies standardisées et en source ouverte pour la communication entre services ou encore la gestion des tâches asynchrones. Cela enlève tout l'intérêt des contributions de la communauté *OpenTracing*, parmi lesquelles on trouve par exemple une instrumentation de la librairie *gRPC* d'appel de procédure à distance. Il faut cependant s'attendre à ce qu'avec l'adoption croissante de telles librairies, il soit plus simple d'instrumenter une application distribuée avec *OpenTracing* et ainsi de pouvoir utiliser notre solution.

Une seconde limitation réside dans la difficulté à appliquer notre solution pour cibler des applications s'exécutant dans des conteneurs Linux. C'est un cas de plus en plus fréquent pour des applications distribuées, mais le traceur *LTTng* ne permet pas aisément de collecter en même temps, et de manière utilisable pour les analyses, une trace noyau sur la machine, et une trace dans l'espace utilisateur pour un conteneur. *LTTng* est cependant en constant développement, et le support des conteneurs devrait être meilleur dans la prochaine version du traceur.

Enfin, les résultats présentés dans l'article montrent un surcoût assez important de notre solution, lorsqu'il s'agit de collecter les traces pour une application traitant un débit élevé de requêtes. La seule solution efficace pour diminuer le surcoût est de diminuer fortement le pourcentage de requêtes tracées par *Jaeger* ; c'est d'ailleurs une situation tout à fait acceptable pour une application en production, qui va malgré l'échantillonnage générer suffisamment de traces intéressantes. En revanche, notre analyse nécessite d'avoir accès non seulement aux

événements de la requête ciblée, mais aussi à ceux correspondant à toutes les requêtes concurrentes. Le cas d'utilisation analysé dans l'article a donc nécessité de collecter les événements correspondant à toutes les requêtes. Afin de concilier nos analyses avec un échantillonnage plus faible des requêtes – et donc un surcoût limité –, il faudrait développer des politiques d'échantillonnage capables de tracer un ensemble de requêtes concurrentes, plutôt que des requêtes prises individuellement.

6.3 Améliorations futures

Ce projet de recherche ouvre des perspectives d'amélioration et des pistes de recherche intéressantes.

Une première piste à explorer est la meilleure intégration de notre solution avec l'écosystème des outils souvent utilisés pour déboguer des applications réparties : les traceurs distribués, les outils de déploiement et les logiciels de monitorage. L'intérêt est de proposer une architecture permettant aux outils de fonctionner en synergie, avec par exemple un support accru par notre solution des politiques d'échantillonnage du traceur distribué, ou encore le déclenchement automatique de prise et d'analyse de traces noyau lorsqu'une anomalie est détectée. Il serait intéressant d'implémenter l'automatisation du déclenchement de la prise de trace et la réception des résultats d'analyse dans des interfaces de monitorage populaires comme Kibana.

La seconde piste à explorer concerne plus directement la qualité des événements collectés ainsi que leur analyse. Nous avons par exemple souligné le manque d'événements associés à la gestion des tâches asynchrones. Il est important, pour assurer la fiabilité des analyses proposées, d'instrumenter des événements tels que l'activation, la mise en file d'attente, le traitement ou la migration entre fils d'exécution d'une requête.

RÉFÉRENCES

[1] Marcos K Aguilera et al. « Performance debugging for distributed systems of black boxes ». In : *ACM SIGOPS Operating Systems Review*. T. 37. 5. ACM. 2003, p. 74–89.

[2] Akshat Aranya, Charles P Wright et Erez Zadok. « Tracefs : A File System to Trace Them All. » In : *FAST*. 2004, p. 129–145.

[3] Dan Ardelean, Amer Diwan et Chandra Erdman. « Performance analysis of cloud applications ». In : *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 2018, p. 405–417.

[4] Mona Attariyan, Michael Chow et Jason Flinn. « X-ray : Automating root-cause diagnosis of performance anomalies in production software ». In : *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 2012, p. 307–320.

[5] Paul Barham et al. « Using Magpie for request extraction and workload modelling. » In : *OSDI*. T. 4. 2004, p. 18–18.

[6] Ali Basiri et al. « Chaos engineering ». In : *IEEE Software* 33.3 (2016), p. 35–41.

[7] Cory Bennett et Ariel Tseitlin. « Chaos monkey released into the wild ». In : *Netflix Blog* (2012).

[8] David Bernstein. « Containers and cloud : From lxc to docker to kubernetes ». In : *IEEE Cloud Computing* 1.3 (2014), p. 81–84.

[9] Ivan Beschastnikh et al. « Leveraging existing instrumentation to automatically infer invariant-constrained models ». In : *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, p. 267–277.

[10] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal et al. « Dynamic Instrumentation of Production Systems. » In : *USENIX Annual Technical Conference, General Track*. 2004, p. 15–28.

[11] Keegan Carruthers-Smith. *GopherCon 2018 - Adventures in Cgo Performance*. GopherCon 2018. 2018. URL : `https://about.sourcegraph.com/go/gophercon-2018-adventures-in-cgo-performance` (visité le 02/07/2019).

[12] Buddhika Chamith et al. « Instruction punning : lightweight instrumentation for x86-64 ». In : *ACM SIGPLAN Notices* 52.6 (2017), p. 320–332.

[13] Anupam Chanda, Alan L Cox et Willy Zwaenepoel. « Whodunit : Transactional profiling for multi-tier applications ». In : *ACM SIGOPS Operating Systems Review*. T. 41. 3. ACM. 2007, p. 17–30.

[14] Haifeng Chen et al. « Boosting the performance of computing systems through adaptive configuration tuning ». In : *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM. 2009, p. 1045–1049.

[15] Michael Chow et al. « The mystery machine : End-to-end performance analysis of large-scale internet services ». In : *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, p. 217–231.

[16] Julien Desfossez, Mathieu Desnoyers et Michel R Dagenais. « Runtime latency detection and analysis ». In : *Software : Practice and Experience* 46.10 (2016), p. 1397–1409.

[17] Mathieu Desnoyers et Michel R Dagenais. « Lockless multi-core high-throughput buffering scheme for kernel tracing. » In : *Operating Systems Review* 46.3 (2012), p. 65–81.

[18] Mathieu Desnoyers et Michel R Dagenais. « The lttng tracer : A low impact performance and behavior monitor for gnu/linux ». In : *OLS (Ottawa Linux Symposium)*. T. 2006. Citeseer, Linux Symposium. 2006, p. 209–224.

[19] DiaMon. *Babeltrace*. n.d. URL : `https://diamon.org/babeltrace/` (visité le 08/05/2019).

[20] DiaMon. *Common Trace Format v1.8.2*. n.d. URL : `https://diamon.org/ctf/` (visité le 08/05/2019).

[21] Francois Doray et Michel Dagenais. « Diagnosing performance variations by comparing multi-level execution traces ». In : *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2017), p. 462–474.

[22] Eclipse. *Trace Compass*. n.d. URL : `https://www.eclipse.org/tracecompass/` (visité le 08/05/2019).

[23] Frank Ch Eigler et Red Hat. « Problem solving with systemtap ». In : *Proc. of the Ottawa Linux Symposium*. 2006, p. 261–268.

[24] Úlfar Erlingsson et al. « Fay : Extensible distributed tracing from kernels to clusters ». In : *ACM Transactions on Computer Systems (TOCS)* 30.4 (2012), p. 13.

[25] Matt Fleming. *A thorough introduction to eBPF*. 2017. URL : `https://lwn.net/Articles/740157/` (visité le 08/05/2019).

[26] Rodrigo Fonseca et al. « X-trace : A pervasive network tracing framework ». In : *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association. 2007, p. 20–20.

[27] Mohamad Gebai et Michel R Dagenais. « Survey and analysis of kernel and userspace tracers on Linux : Design, implementation, and overhead ». In : *ACM Computing Surveys (CSUR)* 51.2 (2018), p. 26.

[28] Amir Reza Ghods. « A study of Linux Perf and slab allocation sub-systems ». Mém.de mast. University of Waterloo, 2016.

[29] Francis Giraldeau et Michel Dagenais. « Wait analysis of distributed systems using kernel tracing ». In : *IEEE Transactions on Parallel and Distributed Systems* 27.8 (2016), p. 2450–2461.

[30] Francis Giraldeau, Julien Desfossez et al. « Recovering system metrics from kernel trace ». In : *Linux Symposium*. T. 109. 2011.

[31] B Gregg. *Strace wow much syscall*. 2014.

[32] Jiamin Huang, Barzan Mozafari et Thomas F Wenisch. « Statistical analysis of latency through semantic profiling ». In : *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, p. 64–79.

[33] Jaeger. *Architecture - Jaeger documentation*. 2019. url : `https://www.jaegertracing.io/docs/1.12/architecture/` (visité le 02/06/2019).

[34] Jaeger. *Jaeger : open-source, end-to-end distributed tracing*. n.d. url : `https://www.jaegertracing.io` (visité le 08/05/2019).

[35] Michael Jeanson. « A follow-up on LTTng container awareness ». Free and Open Source Software Developers' European Meeting. 2019. url : `https://fosdem.org/2019/schedule/event/containers_lttng/` (visité le 14/05/2019).

[36] Jonathan Kaldor et al. « Canopy : An end-to-end performance tracing and analysis system ». In : *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, p. 34–50.

[37] Soila P Kavulya et al. « Draco : Statistical diagnosis of chronic problems in large distributed systems ». In : *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE. 2012, p. 1–12.

[38] Eric Koskinen et John Jannotti. « Borderpatrol : isolating events for black-box tracing ». In : *ACM SIGOPS Operating Systems Review*. T. 42. 4. ACM. 2008, p. 191–203.

[39] Pedro Las-Casas et al. « Weighted Sampling of Execution Traces : Capturing More Needles and Less Hay ». In : *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2018, p. 326–332.

[40] Jacob Leverich et Christos Kozyrakis. « Reconciling high server utilization and sub-millisecond quality-of-service ». In : *Proceedings of the Ninth European Conference on Computer Systems*. ACM. 2014, p. 4.

[41] LTTng. *LTTng v2.10 - LTTng Documentation*. 2018. URL : `https://lttng.org/docs/v2.10` (visité le 02/06/2019).

[42] LTTng. *The lttng-dev Archives*. n.d. URL : `https://lists.lttng.org/pipermail/lttng-dev/` (visité le 08/05/2019).

[43] Shiqing Ma et al. « {MPI} : Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning ». In : *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 2017, p. 1111–1128.

[44] Jonathan Mace, Peter Bodik et al. « Retro : Targeted resource management in multi-tenant distributed systems ». In : *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, p. 589–603.

[45] Jonathan Mace, Ryan Roelke et Rodrigo Fonseca. « Pivot tracing : Dynamic causal monitoring for distributed systems ». In : *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), p. 11.

[46] Steven McCanne et Van Jacobson. « The BSD Packet Filter : A New Architecture for User-level Packet Capture. » In : *USENIX winter*. T. 46. 1993.

[47] Microsoft. *About Event Tracing - Windows applications*. 2018. URL : `https://docs.microsoft.com/en-us/windows/desktop/etw/about-event-tracing` (visité le 08/05/2019).

[48] Alexandre Montplaisir-Gonçalves et al. « State history tree : an incremental disk-based data structure for very large interval data ». In : *2013 International Conference on Social Computing*. IEEE. 2013, p. 716–724.

[49] Karthik Nagaraj, Charles Killian et Jennifer Neville. « Structured comparative analysis of systems logs to diagnose performance problems ». In : *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, p. 26–26.

[50] OpenTracing. *The OpenTracing project*. n.d. URL : `https://opentracing.io` (visité le 08/05/2019).

[51] Cuong Pham et al. « Failure diagnosis for distributed systems using targeted fault injection ». In : *IEEE Transactions on Parallel and Distributed Systems* 28.2 (2017), p. 503–516.

[52] Benjamin Poirier, Robert Roy et Michel Dagenais. « Accurate offline synchronization of distributed traces using kernel-level events ». In : *ACM SIGOPS Operating Systems Review* 44.3 (2010), p. 75–87.

[53] Vara Prasad et al. « Locating system problems using dynamic instrumentation ». In : *2005 Ottawa Linux Symposium*. Citeseer. 2005, p. 49–64.

[54] Charles Reiss et al. « Towards understanding heterogeneous clouds at scale : Google trace analysis ». In : *Intel Science and Technology Center for Cloud Computing, Tech. Rep* 84 (2012).

[55] Patrick Reynolds et al. « Pip : Detecting the Unexpected in Distributed Systems. » In : *NSDI*. T. 6. 2006, p. 9–9.

[56] S Rostedt. « Using the trace event macro ». In : (2017). url : `http://lwn.net/Articles/379903/`.

[57] Raja R Sambasivan et al. « Principled workflow-centric tracing of distributed systems ». In : *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM. 2016, p. 401–414.

[58] Suchakrapani Datt Sharma et Michel Dagenais. « Enhanced userspace and in-kernel trace filtering for production systems ». In : *Journal of Computer Science and Technology* 31.6 (2016), p. 1161–1178.

[59] Harshal Sheth, Andrew Sun et Raja R Sambasivan. « Skua : Extending Distributed-Systems Tracing into the Linux Kernel ». DevConf.US '18. 2018. url : `https://devconfus2018.sched.com/event/FzVg/skua-extending-distributed-systems-tracing-into-the-linux-kernel`.

[60] Yuri Shkuro. *Evolving distributed tracing at Uber engineering*. 2017. url : `https://eng.uber.com/distributed-tracing/` (visité le 08/05/2019).

[61] Benjamin H Sigelman et al. « Dapper, a large-scale distributed systems tracing infrastructure ». In : (2010).

[62] Byung-Chul Tak et al. « vPath : Precise Discovery of Request Processing Paths from Black-Box Observations of Thread and Network Activities. » In : *USENIX Annual technical conference*. 2009.

[63] Xu Zhao et al. « lprof : A non-intrusive request flow profiler for distributed systems ». In : *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, p. 629–644.

[64] Zipkin. *OpenZipkin - A distributed tracing system*. n.d. URL : `https://zipkin.io` (visité le 08/05/2019).